High-performance Computing and Simulation (HCS)
Research Laboratory

Grant Number N00014-96-1-0569
to Florida State University
January - December 1996

"Parallel and Distributed Computing Architectures and Algorithms for Fault-Tolerant Sonar Arrays"

Annual Report #1

Submitted to the:

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5660

February 15, 1997

Attention:
Dr. John Tague
ONR 321(US)

Submitted by:

Alan D. George
PI: Alan D. George, Ph.D.

Assoc. Prof. of ECE and Dir., HCS Research Lab
Dept. of Electrical and Computer Engineering
University of Florida
PO Box 116200, 216 Larsen Hall
Gainesville, FL 32611-6200
Phone: (352)392-5225
e-mail: george@hcs.ufl.edu
(formerly with Florida State University)

Warren A. Rosen adg
PI: Warren A. Rosen, Ph.D.

Professor of ECE
Dept. of Electrical and Computer Engineering
Drexel University
Building 7-411
Philadelphia, PA 19104
Phone: (215)895-6604
e-mail: wrosen@ece.drexel.edu
(formerly with the Naval Air Warfare Center)

Contributing Research Team Members:
A. George, W. Rosen, C. Ih, L. Hopwood, J. Markwell, R. Fogarty,
R. Hardie, A. Holland, and A. Hubbard

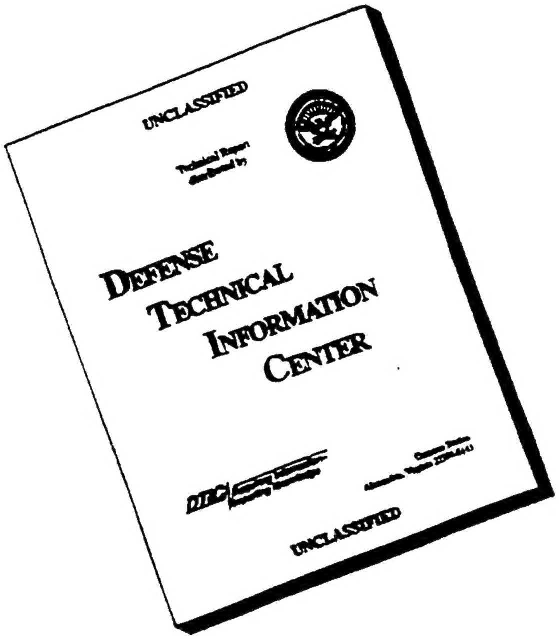

UNCLASSIFIED
Technical Report distributed by
DEFENSE TECHNICAL INFORMATION CENTER
DTIC
UNCLASSIFIED

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Meaning
ACC	Auto- and Cross-Correlation Beamforming
ADC	Analog-to-Digital Converter
AGC	Automatic Gain Control
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
BDE	Block Diagram Editor (in BONeS)
BDN	Bounded-Degree Network
BDS	Basic Delay-and-Sum Beamforming
BF	Beamforming
BONeS	Block Oriented NEtwork Simulator
CC-NUMA	Cache-Coherent Non-Uniform Memory Access
COMA	Cache-Only Memory Access
COTS	Commercial-Off-The-Shelf
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
D&C	Divide And Conquer
DCE	Data Communication Equipment
DFT	Discrete Fourier Transform
DLC (DLL)	Data Link Control (Data Link Layer)
DOA	Direction Of Arrival
DPSA	Distributed Parallel Sonar Array
DQDB	Dual Queue Dual Bus
DRAM	Dynamic RAM
DSE	Data Structure Editor (in BONeS)
DSI	Delay-and-Sum with Interpolation
DT2	Decimation in Time by 2
DTE	Data Terminating Equipment
EDU	Electrical Distribution Unit

FDM	**F**requency-**D**ivision **M**ultiplexing
FFT	**F**ast **F**ourier **T**ransform
FIFO	**F**irst **I**n / **F**irst **O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FTP	**F**ile **T**ransfer **P**rotocol
HCS	**H**igh-performance **C**omputing and **S**imulation Research Laboratory
HDLC	**H**igh-Level **D**ata **L**ink **C**ontrol
GUI	**G**raphical **U**ser **I**nterface
ICN	**I**nter**C**onnection **N**etwork
IEEE	**I**nstitute of **E**lectrical and **E**lectronic **E**ngineers
I/O	**I**nput / **O**utput
IP	**I**nternet **P**rotocol
ISO	**I**nternational **S**tandards **O**rganization
LAN	**L**ocal **A**rea **N**etwork
LCC	**L**ogical **L**ink **C**ontrol
LPF	**L**ow-**P**ass **F**ilter
LSB	**L**east **S**ignificant **B**it
MAC	**M**edium **A**ccess **C**ontrol
MAN	**M**etropolitan **A**rea **N**etwork
MMIC	**M**icrowave **M**onolithic **I**ntegrated **C**ircuit
MPI	**M**essage-**P**assing **I**nterface
MSB	**M**ost **S**ignificant **B**it
MTTF	**M**ean **T**ime **T**o **F**ailure
MUTEX (mutex)	**MUT**ual **EX**clusion
NOW	**N**etwork **O**f **W**orkstations
NUMA	**N**on-**U**niform **M**emory **A**ccess
O/S	**O**perating **S**ystem
OSI	**O**pen **S**ystems **I**nterconnect

PBF	**P**arallel **B**idirectional **F**requency-Domain Beamforming
PBT	**P**arallel **B**idirectional **T**ime-Domain Beamforming
PC	**P**hase **C**enter
PCMCIA	**P**ersonal **C**omputer **M**emory **C**ard **I**nternational **A**ssociation
PDC	**P**arallel and **D**istributed **C**omputing
PLD	**P**rogrammable **L**ogic **D**evice
PNF	**P**arallel **N**etwork-independent **F**requency-Domain Beamforming
PNT	**P**arallel **N**etwork-independent **T**ime-Domain Beamforming
PRAM	**P**arallel **R**andom **A**ccess **M**achine
PRF	**P**arallel **R**ing **F**requency-Domain Beamforming
PRT	**P**arallel **R**ing **T**ime-Domain Beamforming
PUF	**P**arallel **U**nidirectional **F**requency-Domain Beamforming
PUT	**P**arallel **U**nidirectional **T**ime-Domain Beamforming
PVM	**P**arallel **V**irtual **M**achine
RAM	**R**andom **A**ccess **M**emory
RDSA	**R**apidly **D**eployable **S**onar **A**rray
RISC	**R**educed **I**nstruction **S**et **A**rchitecture
SBF	**S**equential **B**idirectional **F**requency-Domain Beamforming
SBT	**S**equential **B**idirectional **T**ime-Domain Beamforming
SCI	**S**calable **C**oherent **I**nterface
SEQFFT	Purely **SEQ**uential **FFT** Beamforming
SM	**S**imulation **M**anager (in BONeS)
SMP	**S**ymmetric **M**ulti**P**rocessing
SRF	**S**equential **R**ing **F**requency-Domain Beamforming
SRT	**S**equential **R**ing **T**ime-Domain Beamforming
SNT	**S**equential **N**etwork-independent **T**ime-Domain Beamforming
SNF	**S**equential **N**etwork-independent **F**requency-Domain Beamforming
SPGA	**S**ystem **P**rogrammable **G**ate **A**rray
SRAM	**S**tatic **RAM**
SUF	**S**equential **U**nidirectional **F**requency-Domain Beamforming
SUT	**S**equential **U**nidirectional **T**ime-Domain Beamforming
TCP	**T**ransmission **C**ontrol **P**rotocol
TDM	**T**ime-**D**ivision **M**ultiplexing

UMA	**U**niform **M**emory **A**ccess
VCSEL	**V**ertical **C**avity **S**urface **E**mitting **L**aser
WAN	**W**ide **A**rea **N**etwork

1. Introduction

Quiet submarine threats and high clutter in the littoral undersea environment demand that higher-gain acoustic sensors be deployed for undersea surveillance. This trend requires the implementation of high-element-count sonar arrays and leads to a corresponding increase in data rate and the associated signal processing. The U.S. Navy is developing a series of low-cost, disposable, battery-powered, and rapidly-deployable sonar arrays for undersea surveillance. The algorithms being mapped to these and other sonar arrays are computationally intensive, particularly when environmentally adaptive for detection and classification. As a result, the processing and dependability requirements placed on the data collector/processor, the monolithic embedded computer required to collect and process sensor data in a real-time fashion, are becoming prohibitive in terms of cost, electrical power, size, weight, etc.

Parallel processing techniques together with advanced networking and distributed computing technologies and architectures can be used to turn the telemetry nodes of these autonomous sonar arrays into processing nodes of a large distributed, parallel processing system for sonar signal processing. This approach holds the potential to eliminate the need for a centralized data collector/processor, reduce the aggregate battery drain, and increase overall system performance, dependability, and versatility.

The RDSA (Rapidly Deployable Sonar Array) architecture will be used as a baseline for purposes of comparison. The target current drain and cost for this array are 22 mA per telemetry node plus an estimated 2 A for the end processor and a cost of $500 per node plus an estimated $10K for a ruggedized, packaged end processor.

Products developed under this effort will include algorithms and performance models for the decomposition and mapping of signal processing applications to linear array or ring multicomputers for sonar arrays, a software-based fine-grain system simulator, a small-scale hardware prototype, and a prototype software system capable of running on both the simulator and the hardware prototype.

1.1 Technical Background

Large autonomous sonar arrays such as the one being developed under the Rapidly Deployable Sonar Array (RDSA) program are designed along a basic "freight train architecture." In this architecture the data taken at each node on the network is loaded onto a train which passes down the "track" to a centralized data collector/processor at the end of the track. The data collector/processor represents a single-point-of-failure for the network, a potential performance bottleneck, and is also a major cost driver. For example, a commercial VME-based special-purpose processor in a ruggedized package costs about $10,000 and draws about 7 A of current. A future PCMCIA-based version of this processor might draw as little as 2 A but this would still require on the order of 180 lithium C-cell batteries for a 30-day mission. The present program grew out of an attempt to use advanced parallel and distributed processing techniques and the high bandwidth and low latency of fiber optics to eliminate the centralized data collector/processor and replace it with a processing architecture in which each telemetry node of the network represents a processing element of a parallel processor, essentially turning the array itself into a distributed signal processing machine.

The research for this project is an adaptation of the currently operating sonar array. This array was assumed to contain an arbitrary number of single transducer elements strung together in a linear fashion. The RDSA (Rapidly Deployable Sonar Array) prototype is shown below as Figure 1.1.1. The "dumb" nodes do not contain a microprocessor but do contain some electronics for the network protocol and an analog-to-digital converter to sample the data from the transducer.

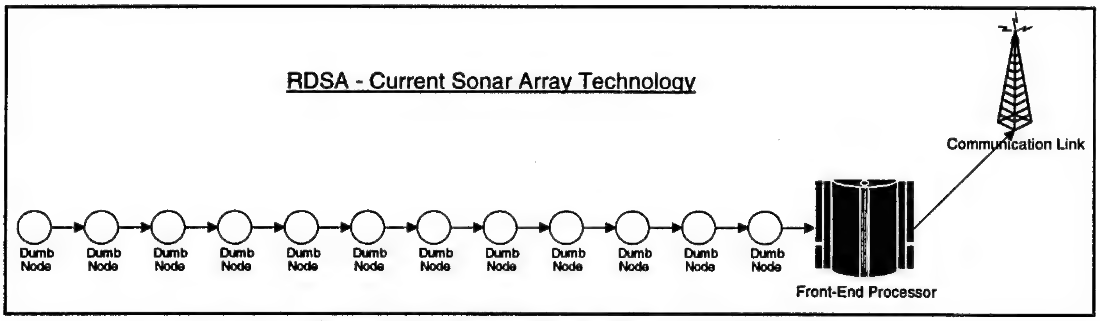


Figure 1.1.1 : Current Technology, The RDSA

Current technology uses a large front-end processor, many dumb nodes, and a communication link. The front-end processor is a single point of failure.

The parallel prototype uses the same physical arrangement although the network's logical topology may be more than a simple unidirectional network. Figure 1.1.2 shows 7 nodes connected via a bidirectional array network. The formal name for this project, "Parallel and Distributed Computing Architectures and Algorithms for Fault-Tolerant Sonar Arrays," well defines the nature of the deliverable intended. Parallel means that the decomposition of the beamform is mapped across many processors. Distributed means that the processors communicate via message passing, as opposed to shared memory. The acronym DPSA or Distributed Parallel Sonar Array is used to refer to the new parallel, distributed, and fault-tolerant sonar array computer under research and development.

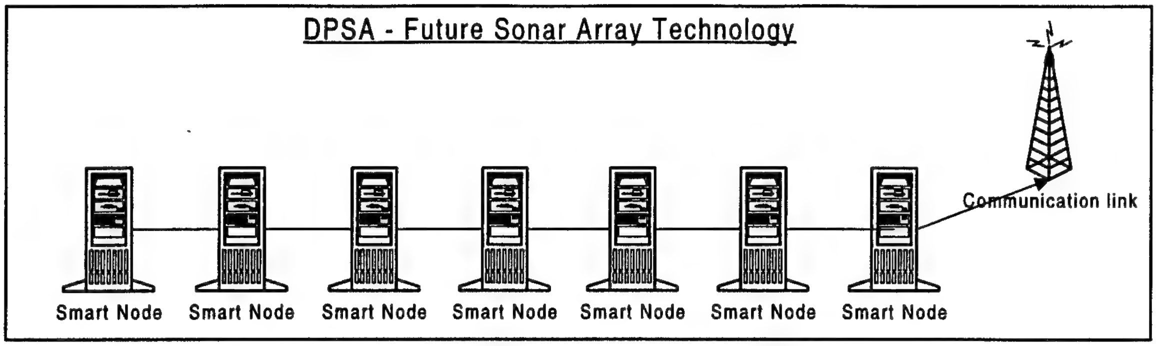


Figure 1.1.2 : Future Technology, The DPSA

The DPSA is composed of smart nodes which distribute the processing, provide fault-tolerance by avoiding a single point of failure, and lower cost by eliminating an expensive front end.

The architecture of a "smart node" is assumed to have a single transducer. The basic layout of such a device may be something like that shown below in Figure 1.1.3. Each node has one main processor for beamform algorithms, a RAM device, an A/D converter, a clock-recovery chip, an optical receiver, a processing element, either ASIC or COTS for the network protocol, a battery, and other various interface/power components. In some cases the complexity of the circuit may increase such as in the case of a bidirectional array in which there would be two optical transceivers, two optical bypasses, and two clock-recovery devices.

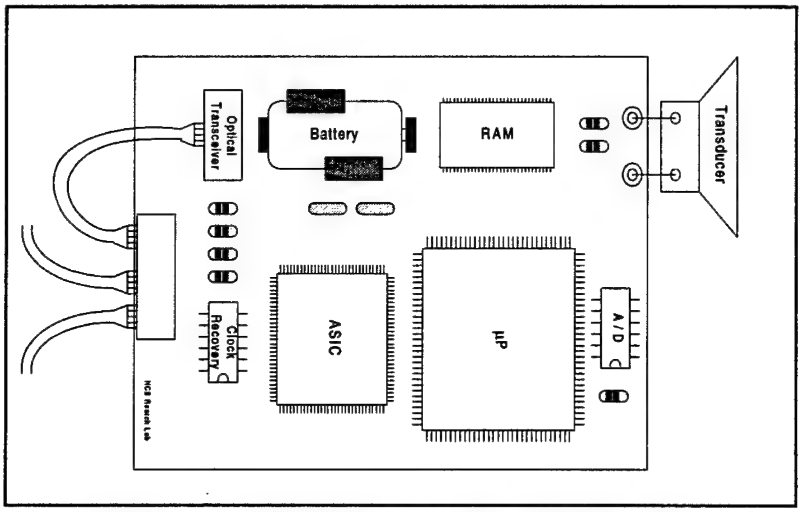


Figure 1.1.3 : Smart Node, The Distributed Parallel Node

The generic figure above represents the future technology of sonar array processing: the Distributed Parallel Node.

In FY96 progress has been made in several areas. First, an analysis of the effect of node outage on network reliability was conducted. This analysis indicated that network reliability could be greatly enhanced by using an optical bypass switch capable of bypassing at least two successive failed nodes. A switch of this type is currently being developed under the SBIR program. Second, a survey of low-cost, low-power networking components was started. So far laser diodes, RAMs, ADCs, and high-power batteries have been investigated and analyses of low-power clock recovery circuits and plastic fiber optic cable is currently underway. When completed this survey will be used to determine the optimal network topology and system architecture. Third, preliminary parallel decompositions of several standard beamforming algorithms have been performed, including delay-and-sum, delay-and-sum with interpolation, and FFT. In addition, an autocorrelation-based algorithm has been developed which takes advantage of the unique architectural features of the network to provide simple, low-power processing of the acoustic data. Algorithms and programs for the sequential versions of these beamforming techniques have been developed in Matlab, C, and MPI to form a baseline by which parallel algorithms and software will be measured. Fourth, a fine-grain model of the baseline freight train protocol has been developed using the Block-Oriented Network Simulator (BONeS) tool, and models for candidate network architectures are under development for unidirectional ring and bidirectional linear array topologies.

1.2 Technical Issues

The technical issues involved with the design and development of parallel and distributed computing architectures and algorithms for fault-tolerant sonar arrays include unsolved problems in four major areas. These interrelated areas are: Topology, Architecture, and Protocol; Algorithm Decomposition, Partitioning, and Mapping; Fault Tolerance; and Simulator Tools.

1.2.1 Topology, Architecture, and Protocol

Given that all technical issues are driven by the network topology and, node, processor, and network protocol design, the development of an efficient and effective architecture and set of protocols for this network-based multicomputer system for large sonar arrays represents a number of key technical challenges. Many important multicomputer architecture considerations must be addressed in terms of speed, cost, weight, power, and reliability for each node. These include the interconnect architecture, processor architecture, memory architecture, I/O architecture, and the proper hardware selection of components associated with each of these critical elements. Architectural components include circuits and devices for sampling, filtering, processing, communication, flow control, and clock recovery, and their design or selection poses a number of unresolved problems to balance performance and reliability with power, weight, size, and cost.

Network topology is an issue because it is a major cost driver and impacts both the algorithm decomposition and the protocol. The network may be linear and unidirectional (i.e. the baseline RDSA topology), linear and bidirectional, ring-like or a hybrid combination of these. In a bidirectional topology, data is passed in two directions, either over separate fibers or through the same fiber over different wavelengths. In a ring topology, the last node in the array is connected to the first to form a ring-like structure. The choice of topology is strongly related to the cost and power requirements of the networking components used, in particular the optical transmitter, receiver, and clock recovery. A linear unidirectional network requires the smallest number of components but cannot support many parallel algorithms and does not support a high degree of fault tolerance. A linear bidirectional network is extremely robust and will support any parallel algorithm but requires twice the number of optical link components and a more complex protocol. The ring topology supports all algorithms but requires a long cable run and is the least fault tolerant. Newly developed plastic fiber represents an economical alternative to glass but has high attenuation, and therefore will not support a physically large ring. The choice of a topology will require a trade study of both the component characteristics and protocol and algorithm impact.

Network protocol is another key issue because currently available protocols are prohibitively power hungry. For example, a typical commercial FDDI chipset draws 400 mA. A custom protocol must be extremely efficient yet support real-time performance. A number of unresolved problems must be addressed including message-routing primitives and schemes, network flow-control strategies, deadlock avoidance, and virtual channels. For all these unresolved problems in topology, architecture, and protocol, success is measured in terms of hardware and protocol flexibility, versatility, expandability, scalability, cost, weight, power, size, etc. as compared to the baseline RDSA sonar array.

1.2.2 Algorithm Decomposition, Partitioning, and Mapping

The design and development of novel, new parallel and distributed algorithms for large sonar array beamforming applications, and the partitioning and mapping of these algorithms onto candidate network topologies, architectures, and protocols, is one of the most pivotal technical issues in this project. This is an area not addressed by conventional literature in the field. Rapidly evolving theoretical and algorithmic developments in the area of digital signal processing will be the basis for program decomposition onto candidate multicomputer architectures for sonar arrays. These arrays may potentially exhibit multibeam and multifrequency operation, increased acoustic sensitivity, digital beamforming, and adaptive processing. When designing and developing multicomputer architectures and protocols, the degree of potential matching between the architecture and the application algorithm must be analyzed via scalability studies. For different architecture and algorithm pairs, the analysis may often lead to very different conclusions. Therefore, another critical technical issue is scalability studies for the determination of the most optimal combinations of the candidate architectures with the most time-critical algorithms partitioned in various ways. Key problems in program partitioning and scheduling for distributed, parallel computers include granularity, latency, grain packing, and scheduling. Some of the parameters involved in this process include machine size (i.e. the number of nodes), clock rate or machine cycle, problem size, parallel execution time in terms of problem and machine size, I/O demand, memory capacity, communication overhead, cost, weight, power, size, etc. A number of decomposition techniques must be considered, including domain decomposition, control decomposition, object decomposition, and layer decomposition techniques, and each involves message-passing programming and performance tuning. Success is measured in terms of algorithm and software complexity, distributed/parallel performance, architectural correlation, flexibility, versatility, expandability, scalability, maintainability, etc. as compared to the baseline sequential sonar array.

1.2.3 Fault Tolerance

Another critical technical issue that must be addressed for this project is the determination of the most effective and yet efficient architecture and self-healing algorithm design strategies to improve the reliability and mission time of a large sonar array system. Like parallel and distributed algorithms for large sonar array applications, conventional literature in fault-tolerant computing does not address the unique requirements posed for these types of systems. Given the environment in which these systems will operate, component failure during the desired mission time is a major concern, repair is not an option, and it is imperative that the loss of individual processors, memory units, I/O units, interface circuits, and network links does not impede the ability of the system to perform with graceful degradation. A key element of the fault tolerance design studies conducted must be how best to improve fault coverage, reliability, and mission time while keeping low the power, weight, and cost factors. This balance is a critical unresolved problem for this or any system. Success is measured in terms of system reliability, mission time, graceful degradation, and the price paid for these in terms of power, weight, size, and cost as compared to the baseline sequential sonar array.

1.2.4 Simulator Tools

Given the increasingly high costs associated with the development of system prototypes, especially those based on complex systems designed for distributed and parallel processing, modeling and simulation techniques and tools have rapidly become a critical enabling technology and from them has arisen the field of rapid virtual prototyping. Based on Monte Carlo, Markov, Petri net, and other techniques, such simulation-based prototyping methods hold the promise to revolutionize the way in which computing systems such as those proposed for large sonar arrays can be efficiently and effectively designed. However, given the lack of appropriate simulator tools for the algorithms and architectures associated with this effort, another key technical issue is the development and exploitation of new simulator tools. For example, while no fine-grain modeling and simulation tools exist which alone are suitable for sonar array multicomputer architecture development, protocol development, fault tolerance, and program and algorithm decomposition and mapping, there are several tools which may be leveraged, adapted, developed, and integrated via simulator tools developed by this project and the optimum methods for this are themselves key technical issues in terms of both performance and dependability analysis. Success is measured in terms of fidelity, flexibility, versatility, expandability, scalability, transition potential, and the simulator's inherent ability to demonstrate feasibility and assist in quantifying performance and dependability improvements versus sequential and candidate baselines.

1.3 Technical Approach

The technical issues will be addressed by a three-phase approach to the design and development of parallel and distributed architectures and algorithms for fault-tolerant sonar arrays. Together, these phases will work to prove and demonstrate how advanced techniques in parallel and distributed processing, computer networks, and fault-tolerant computing can be effectively and efficiently employed to construct next-generation sonar arrays with less cost and size and a greater degree of dependability, performance, and versatility.

In the first phase, tasks will concentrate on the study, design, and analysis of the fundamental components for these advanced sonar arrays. An interactive investigation of the interdependent areas of topology, architecture, protocol, and algorithms will be conducted. The results of this investigation will include the optimum candidate network topology, architecture, hardware components, and interface protocols for the system architecture and a set of fundamental decomposition techniques and parallel algorithms for a representative set of time-domain and frequency-domain beamforming methods.

In the second phase, tasks will concentrate on the critical steps to bridge from the results of fundamental studies in the first phase to the eventual goal of a small hardware prototype and will include the development of the preliminary software system for the advanced sonar array, the design of a strawman node circuit, and the development of a suite of simulation tools which support the design and analysis of both system performance and dependability. Parallel programs will be developed with inherent granularity knobs based on the results of algorithm decomposition and partitioning in the first phase. Self-healing extensions of the selected algorithms will be developed and simulated in this phase. A strawman hardware node design will be conducted to determine estimated gate counts and power requirements. Finally and concurrently, a suite of simulation tools will be developed and integrated to support the rapid virtual prototyping of topology, architecture, protocol, and algorithm selections made in the first phase.

In the third phase, the emphasis is on the development, implementation, and demonstration of a small, laboratory-based hardware prototype with its own software system which will be fabricated, used to verify and validate the simulation results, and in so doing demonstrate and better quantify the inherent advantages of the novel approach employed in its design. Critical factors in the evaluation of the system will center on quantitative measurements in the areas of performance and dependability, including computational speed, efficiency, and precision, reliability, cost, weight, power, size, and mission time, as well as qualitative measurements related to system flexibility, versatility, expandability, scalability, etc. All of the measurements will be compared with the baseline RDSA system architecture and sequential beamforming algorithms to measure the degree of success.

1.4 Technical Results

An analysis of the effects of random node failures on the beam power pattern has been completed. The biggest effect of node failure is in the side lobe pattern rather than in the main beam. The results show that a substantial number of nodes may fail before the main beam power is significantly degraded. Therefore, it is essential to be able to bypass optically one or more failed nodes.

A survey of low-cost, low-power networking and processing components is nearing completion. So far laser diodes, RAMs, analog-to-digital converters, and high-power batteries have been investigated and analyses of low-power clock recovery circuits, microprocessors, and plastic fiber optic cable are underway. Initial results indicate that not only is it possible to design and implement the elements in this distributed processing sonar array via low-power custom devices, but based on cutting-edge technologies including Li/SO$_2$ batteries and low-power integrated circuits it may indeed be possible to attain the 30-day mission time requirement largely with COTS components. For example, processors are now emerging capable of less than 1 mA/MHz at 3V, low-power 3V DRAM devices will soon attain current drains as low as 9 mA for cycle times less than 1000 ns, and Li/SO$_2$ batteries now have an actual average capacity of over 200 Watt-hours/kilogram. Vertical cavity surface emitting lasers (VCSELs) are becoming available which offer sub-milliamp threshold currents. At present, they are expensive due to their low yield; however, they are the subject of extensive research due to their excellent temperature properties, and it is likely that this will result in higher yields and lower cost in the future. Should these initial design estimates prove true in further experiments, this approach will dramatically increase the flexibility and versatility of the architecture, algorithms, and protocols under development allowing such arrays one day to support autonomously a wide variety of beamforming, detection, and classification mechanisms without changing the hardware.

Given that all technical issues are driven by the network topology design, the network protocol design, and the node processor, the development of an efficient and effective architecture and set of protocols for this network-based multicomputer system for autonomous sonar arrays represents a number of key technical challenges. Many important multicomputer architecture considerations must be addressed in terms of speed, cost, weight, power, and reliability for each node. To address these considerations, network architectures based on unidirectional ring, unidirectional linear array, insertion ring, and bidirectional linear array are undergoing modeling, simulation, and analysis in fine-grain model form to study their inherent performance, complexity, and dependability characteristics. In addition, a fine-grain baseline model for the RDSA unidirectional freight-train network has been constructed and simulated, and will be used to compare the performance of proposed protocols and topologies. Initial results have been successful in reducing the complexity of existing network protocols by trimming down standard components in the OSI protocol stack to reduce power and cost factors.

The design and development of novel parallel and distributed algorithms for large sonar array beamforming applications, and the partitioning and mapping of these algorithms onto candidate network topologies, architectures, and protocols, is one of the most pivotal technical issues in this project. To attack this task in the project, a survey of all decomposition methods thus far in the field has been conducted from which a single comprehensive approach is being devised for this application area. By preparing a thorough, architecture-independent decomposition of candidate algorithms, the communication and computation requirements of each method can be thoroughly examined for application to each candidate architecture, at which time the most suitable parallel decomposed algorithms can be chosen for further experimentation. Meanwhile, a collection of conventional and non-conventional beamforming algorithms has been constructed. Preliminary parallel decompositions of several standard beamforming algorithms have been performed, including delay-and-sum, delay-and-sum with interpolation, and FFT. In addition, an autocorrelation-based algorithm has been developed which takes advantage of the unique architectural features of the network to provide simple, low-power processing of the acoustic data. Algorithms and programs for the sequential version of these beamforming techniques have been developed in MATLAB and C sequential code and MPI (Message-Passing Interface) skeleton parallel code to form a baseline by which true parallel algorithms and software will be measured. Initial results from performance experiments with some parallel programs developed to date indicate the potential for near-linear speedup and a high degree of parallel efficiency when properly mapped to network architectures similar to those candidates under consideration. A prototype X-Windows simulator has been constructed which supports multithreaded MPI parallel code which can be mapped to several networks, workstations, and processors in the parallel processing testbed developed for this project. Preliminary tests with this simulator have been successful in identifying and measuring the strengths and weaknesses of parallel programs being studied versus their baseline counterparts.

1.5 Tasks

Task 1. Topology, Architecture, and Protocol Development

An analysis and selection/development of a candidate network topology, architecture, and interface protocol will be conducted. Hardware will include optical transceiver components, clock recovery circuits, batteries, protocol ASICs, and node processing elements. A high-level cost and power model will be developed based on an analysis of fiber run lengths, approximate gate counts, and low-power optics.

Task 2. Algorithm Development and Modeling

A selection/development, analysis, and decomposition of digital signal processing algorithms with respect to their suitability in ring or linear-array multicomputer architectures for sonar arrays will be performed and an analysis and selection of parallel programming tools will be carried out. Representative algorithms from both time-domain and frequency-domain beamforming will be modeled and analyzed using analytical and experimental techniques so that tradeoff analyses can be conducted and the selection of algorithm and architecture features can be made. Modeling and analysis will be performed with respect to decomposition method and degree of granularity.

Task 3. Simulator Development

The results of the modeling and development process will be used to construct a fine-grain, high-fidelity simulator suite capable of accurately rendering node, network, and system behavior. This simulator suite will represent the performance of the basic RDSA system as the baseline as well as the new distributed, fault-tolerant architecture and parallel, self-healing programs which are the emphasis of this project. The simulator suite will be equipped with a graphical user interface and will make it possible to gauge the potential performance of candidate architectures and algorithms and measure relative success and failure to achieve specific goals. The simulator suite will be expandable to include both low-performance and high-performance architectures for different mission requirements and will serve as the platform on which the preliminary decomposition, mapping, and tuning of the prototype software system will be developed.

Task 4. Preliminary Software System Development

Based on the results of algorithm decomposition and partitioning, a series of preliminary parallel programs will be constructed with inherent granularity knobs and portability for mapping to various distributed/parallel architectures. These algorithms and programs will be mapped and tuned for the linear array and ring topologies under consideration. Parallel programs shall be extended to include basic self-healing mechanisms. A series of test and evaluation experiments will be conducted in order to the determine the tradeoffs implied by the matching of these parallel architectures and algorithms.

Task 5. Strawman Node Circuit Design

A strawman functional circuit design will be conducted and a high-level gate count and node power requirements will be determined based on a COTS and ASIC foundry analysis and power consumption evaluation. A functional processor/interface circuit will also be designed. The results of the simulations together with the strawman design will be used to verify that system performance, fault tolerance, and power requirements are met, and the process will be iterated, if necessary.

Task 6. Hardware Prototype Development

An operational small-scale hardware prototype of the network will be constructed that will support the detailed decomposition, mapping, and tuning of the prototype software system and verify simulation results.

Task 7. Detailed Software System Development

The preliminary parallel and self-healing programs will be extended to form the development of a prototype software system for this distributed signal processing machine. These programs will be mapped to the hardware prototype, tuned for performance and dependability, and a series of test and evaluation experiments will be conducted to ascertain and demonstrate the superior capabilities of this hardware and software system as compared to their baselines.

2. Background

2.1 HCS Research Lab Resources

The *HCS Research Laboratory* has a variety of hardware and software resources suitable for any type of simulation or parallel program development. HCS operates several parallel processing testbeds which are used to verify parallel speedup and algorithm validity. The lab also has a variety of parallel programming tools designed for these testbeds. The testbeds are used to execute simulations using tools developed in-house and commercially.

2.1.1 Hardware

The HCS Lab operates three cluster testbeds using six different interconnect technologies. The cluster testbeds use SPARC-based workstations including ten SPARCstation-5/85 workstations, eight dual-processor SMP (symmetric multiprocessor) SPARCstation-20/85 workstations, and eight ULTRAstation-1/170 workstations. The cluster interconnection networks are described below:

Ethernet. All machines have 10-Mbps Ethernet implemented either as 10Base2 or 10BaseT. Support for 100-Mbps Ethernet is under consideration to form a natural comparison to the ATM network.

Asynchronous Transfer Mode. All workstations are equipped with 155-Mbps FORE Systems ATM adapters. ATM is used for all regular UNIX traffic as well as any interprocessor communications. The clusters use three different FORE ASX-200 switches and a FORE PowerHub 7000 for Internet routing.

Myrinet. The ULTRAstation cluster is connected by Myrinet, which is a 1.28-Gbps switched network. The switch is a non-blocking, 8-way crossbar switch. The adapters and switch are built by Myricom and their drivers support either raw interfacing or Ethernet Encapsulated Packets for TCP/IP traffic.

1-Gbps Scalable Coherent Interface. The SPARCstation-20/85 SMP cluster is connected via switched 1-Gbps SCI network using Dolphin SCI/Sbus-1 adapters. The SCI network can be configured either as a single 8-node unidirectional ring or four ringlets connected via a 4-way switch. The switch is also from Dolphin Interconnect Solutions and uses a 3.2-Gbps shared bus as the switch fabric.

1.6-Gbps Scalable Coherent Interface. Four of the Ultra SPARC workstations form a second SCI ringlet using Dolphin SCI/Sbus-2b adapters which operate at 1.6-Gbps. These are second generation SCI adapters designed for high-performance workstations.

Fibre Channel. Two of the SPARCstation-5/85 workstations are connected with a 1.062-Gbps Fibre Channel loop. Fibre Channel is being investigated as a possible backbone architecture for medium-sized networks as an alternative for high-speed I/O networks.

2.1.2 Software

In order to implement the various programs and simulations in this research, a number of software tools were utilized. All work was done on SPARCstation-20s running Solaris 2.5.1 and SPARCstation-5s running Solaris 2.4. Programs were compiled using GNU's C and C++ compilers, versions 2.6.3 and 2.7.2, respectively. Also used was the Sun C compiler, version 4.0.

Parallel programs were run over Ethernet, SCI, and ATM. The Ethernet and ATM programs ran over the entire sets of SPARCstations and used MPICH from Argonne National Laboratory and Mississippi State University. MPICH offers an MPI programming interface for use with workstation clusters that use TCP/IP as the communication protocol. The SCI programs ran over the SPARCstation-20 SMP testbed equipped with Dolphin SCI/Sbus-1 cards and used MPISCI by Parallab and the University of Bergen. MPISCI is specially designed to use the raw transport interface to the SCI adapters.

Other software tools used include network modeling and simulation tools, mathematical prototyping tools, and UNIX system programming tools:

Block-Oriented Network Simulator. BONeS, released by the Alta Group of Cadence Design Systems, Inc., is a fine-grain network modeling and simulation tool that has been extensively used in this project for high-fidelity simulations. All network simulations in this research use BONeS. Previous work using BONeS includes an SCI emulation model, SCI real-time protocol modeling, and modeling and simulation of the Navy TACAMO project.

MATLAB. MATLAB by MathWorks, Inc. is a PC-based mathematical modeling tool that was used to verify beamform algorithm correctness.

Altia. Altia Design Release 1.40 by Altia, Inc. is a graphical user interface (GUI) development program for X-windows based UNIX workstations. Altia simplifies user interfaces with efficient and useful graphical controls.

GNUPLOT. GNUPLOT version 3.5 is released from the GNU software consortium which supplies free UNIX development software. GNUPLOT creates charts and plots of scientific data on UNIX workstations.

Other software tools, including HCS Threads and the text-based runtime system, were developed in-house at the *HCS Research Laboratory*.

2.2 Networking

The Open Systems Interconnect (OSI), a standard produced by the International Standards Organization (ISO), is a well defined hierarchy of network layers. Each of the systems modeled in this project use this standard to some degree. They may, however, not include each of the seven layers defined. Figure 2.2.1 below shows the basic protocol stack beginning with the physical layer and ending with the application layer.

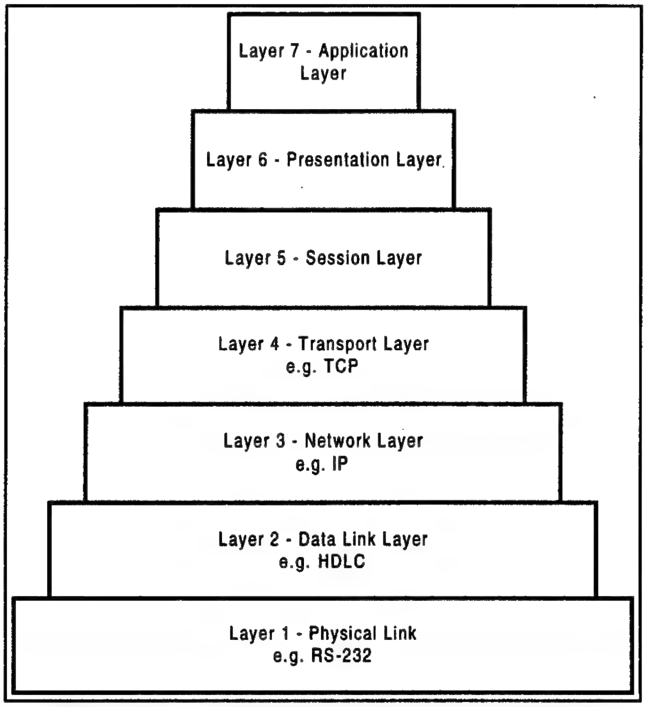

Layer 7 - Application Layer
Layer 6 - Presentation Layer
Layer 5 - Session Layer
Layer 4 - Transport Layer e.g. TCP
Layer 3 - Network Layer e.g. IP
Layer 2 - Data Link Layer e.g. HDLC
Layer 1 - Physical Link e.g. RS-232

Figure 2.2.1 : Seven-layer OSI network architecture
This figure depicts the seven-layer OSI model as defined by the International Standards Organization.

The layers begin at the bottom with the Physical Layer. This layer provides the "virtual bit pipe" between two nodes and thus physically must make the connection [BERT92]. Two common examples of physical layers are the RS-232 serial transmission protocol and the X.21 serial transmission protocol used in the European X.25 standard. The physical layer may also be called a modem or a DCE (Data Communication Equipment) since modulation schemes will typically be used at this level. The physical layer communicates along a physical medium such as copper or fiber. It also communicates with the higher data-link control layer or DTE (Data Terminating Equipment).

Two encoding techniques are being considered for use at the physical level: the S-20 encoding scheme and a 12B16B block encoding scheme. Encoding schemes are used to remove the DC component of the signal, insure a sufficient number of transitions for proper functioning of the clock recovery circuit, and, in some cases, provide a level of error detection. Balancing is achieved by sending an equal number of binary 1's and 0's over a given block of data or time frame. In addition, by using a code with no DC component, an AC-coupled receiver may be used for greater receiver sensitivity. The block encoding scheme is a theoretically simple concept. A data word of *m* bits is encoded as a word of *n* bits, where *n>m*. Since there are many more possible combinations of words of *n* bits than with *m* bits, only the combinations which represent a balanced word are used. The 12B16B block code is a custom-designed code designed at the Navy Air Warfare Center. It is essentially two 5B6B codes and a 2B4B block code. Below in Table 2.2.1 are the values associated with each binary number 0 to 31 for 5B6B and 0 to 3 for the 2B4B code. Notice that for each 5B6B code there are 32 combinations which are illegal (since there are a total of 64 combinations in 6 bits). Therefore, if the receiver collects data which has an illegal code, it immediately detects it. So besides DC balancing and providing sufficient transitions, it also serves as a simple error detection technique, although double errors many times will not be detected, just as in the case of parity checking. The 12B16B code described above is a variation of IBM's Fibre Channel protocol. The FC-1 layer defines an 8B10B implementation. The 8B10B code is simply made up of a 5B6B and a 3B4B code.

0 00000	010111	12 01100	011001	24 11000	110001
1 00001	011011	13 01101	011010	25 11001	110010
2 00010	011101	14 01110	011100	26 11010	110100
3 00011	000111	15 01111	011110	27 11011	110110
4 00100	101101	16 10000	101011	28 11100	111000
5 00101	001011	17 10001	100011	29 11101	111010
6 00110	001101	18 10010	100101	30 11110	111001
7 00111	001110	19 10011	100110	31 11111	100111
8 01000	110011	20 10100	101001	0 00	0101
9 01001	010011	21 10101	101010	1 01	0110
10 01010	010101	22 10110	101100	2 10	1001
11 01011	010110	23 10111	101110	3 11	1010

Table 2.2.1 : 5B6B, 2B4B

This table shows the values associated with each binary number 0 to 31 for 5B6B and 0 to 3 for the 2B4B code.

S-20 is a serial encoding technique which is implemented in the Scalable Coherent Interface (SCI) protocol. It basically adds a 4-bit binary number to the end of a 16-bit binary number to correct the current DC offset. S-20 has the added bonus that it guarantees that the third from the last bit and the second from the last bit will always have a state transition. This is useful for clock synchronization purposes.

The Data Link Layer houses two sublayers which are fundamental to the network's utility: the Medium Access Control (MAC) and the Logical Link Control (LLC) [STAL94]. The MAC does what it implies: it provides medium access control, or contends for the resources of the network. Implementation for this control may be as simple as 1) a "free-for-all," random access in which collisions are likely to occur, or a complicated scheme such as 2) a token-passing protocol or 3) a reservation protocol in which nodes may have access to the network via a First-In–First-Out (FIFO) structure. These three implementations are called Contention, Round Robin, and Reservation, respectively. The most widely used and recognized examples of MAC protocols are CSMA/CD (Ethernet) and IEEE 802.5 Token Ring standard.

Contention is the simplest of all MAC protocols, and it is somewhat chaotic in nature or "free-for-all." In this structure any node may at any time place its information on the bus with the hope that it will not find a collision. A collision occurs if more than one node tries to send information at the same time. The problem with this case is that high utilization of the line completely breaks down the network. Aloha, an early random contention-based protocol, provides for a maximum utilization of 18%. Other contention types do a much better job of utilizing the network. For instance, Carrier Sense Multiple Access with Collision Detection (CSMA/CD) senses whether or not the network is currently being accessed and a node can determine a time when it can probably safely place something on the network. Although collisions are still likely to occur, a much higher utilization may be achieved with this scenario. This utilization is a function of the probability that any station will transmit at any time, the propagation delay of the line, and the number of nodes on the network.

A second type of MAC is the Round Robin or Token-passing protocol. A token is a key which is passed from node to node which allows a node to access the network for a specified task. If a node has nothing to send it simply passes the token to the next node. A token-passing protocol may be physically implemented in the structure of a ring (i.e. a token ring) or theoretically made into a ring (i.e. a token bus). A token bus may be implemented by having the network assign and remember a token-passing loop. In other words, each node would be told to pass the token to a predetermined destination node. The advantage of this type of protocol is its inherent high utilization. If the normalized delay of the line is of low order, the token bus or ring can achieve utilization of above 90%, although this would be very unusual. One relationship of the token protocol which is not as intuitive is that more nodes transmitting at any given time results in higher utilization. We can immediately realize this relationship if we consider the time the network spends passing the token to nodes which do not need to transmit any data. This is opposite to a contention-type network, in which the higher the number of nodes transmitting, the lower the utilization of the line.

The third type of MAC is the reservation protocol. A common example of this type is the Dual Queue Dual Bus (DQDB) structure. This MAC is dependent on the physical nature of the network. A DQDB utilizes a bidirectional bus, although another reservation MAC could utilize a ring topology. A bidirectional bus is a much more ambitious task than a simple unidirectional bus, and access to the network is a much more involved process. In this case we can explore the well known DQDB network protocol. This type of network is based on a reservation request and a FIFO structure. Nodes of a DQDB may send requests upstream and pass information downstream. It is therefore the task of the nodes at the end of each side of the bus to maintain a FIFO structure and to send packets downstream. These packets have a data field and a control field. The control field may be used for each node to send a packet request to the upstream node. The end nodes may also place their own data packet on the network, but they must also place themselves on the FIFO structure and wait their turn. Nodes which are not at the endpoints also maintain a FIFO structure and place themselves on their FIFO. If an empty packet comes down the stream to a node and that node is at the top of the scheduling FIFO queue, then the node may transmit its data. The

DQDB network also has a relatively high utilization in all cases. If a few nodes or all the nodes want to transmit, the network is constantly utilized. This technique is the most complicated of all the protocols, but the advantages of using this type of system justifies these complications.

The Logical Link Control provides the interface service to the higher-level network layers. This service may be acknowledged and non-acknowledged service. Acknowledged service requires that each receiver send an acknowledge back to the source to guarantee transmission. Non-acknowledged data may be supported in transmissions which can drop packets in non-critical real time applications such as video or audio. With unacknowledged LLC, guaranteed transmissions can be serviced in a higher layer such as the transport layer. The sublayers of the DLL are shown below in Figure 2.2.2 with their respective duties.

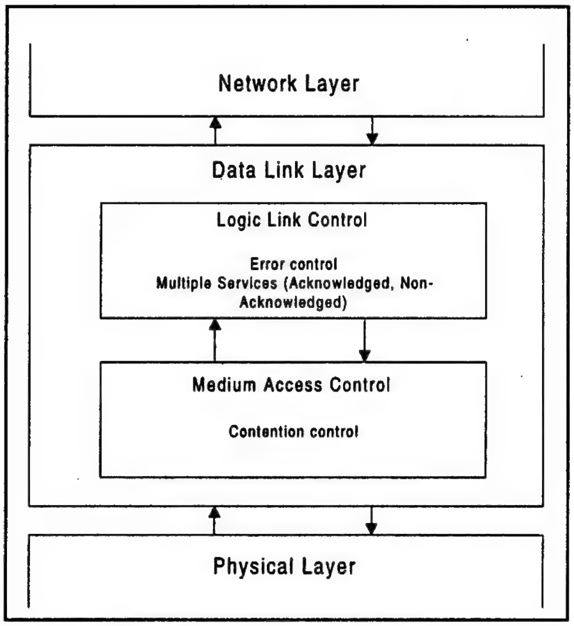


Figure 2.2.2 : Data Link Layer

The data link layer is split into two sublayers whose functions are divided into error control/multiple services and contention control.

The Network Layer provides control or system management with congestion control, routing decisions, link failures, and other global utilities. Routing decisions are based on the service required by the user of the network, which is typically the transport layer. Two services provided are virtual-circuit and datagram service, otherwise known as connection-oriented and connectionless, respectively. A datagram service protocol routes packets by sending each packet independently through the network. Since packets may arrive at the destination out of order, the protocol typically needs to reorder the packets correctly. A virtual-circuit connection sets up a connection between two nodes through any number of hops. Each packet from the node is then sent along this path. Note that virtual circuit is different from circuit-switched technologies. It is still packet-based, and the path is Time-Division Multiplexed (TDM) with other packets [STAL94]; whereas circuit switched technology is not packet-based but still may be multiplexed with

Frequency-Division Multiplexing (FDM). Congestion control or flow control is a means of routing packets around congestion and is another critical function of the network layer. The most widely known example of the network layer is the Internet Protocol (IP) of the TCP/IP standard.

The fourth layer, the Transport Layer, has many functions of which some (but not necessarily all) are supported in every implementation. These include segmenting and re-assembling incoming data streams from the Session Layer, multiplexing low-rate sessions, ensuring reliability, and controlling end-to-end flow. Segmenting data streams requires the transport layer to break apart data structures which are larger than the network layer can handle. The receiving node must reassemble the packets before sending them to the session layer. If a datagram service is used, the transport layer may be used to reassemble the packets in the correct order. Datagram services do not guarantee ordered delivery as do virtual-circuit services. An example of a transport layer is the Transmission Control Protocol (TCP) of the TCP/IP standard.

The session layer performs various jobs such as control of access rights and also sets up sessions between two entities. Support for access rights is a security feature which allows the exchange of records on LANs and MANs. The presentation layer is responsible for data compression and encryption. Although historically used primarily by the military, new high-rate data applications like streamed video and audio are using this layer to compress data before transmitting. The last layer, the application layer, simply contains all of the applications which may require a network connection. These include file transfer protocol (FTP), telnet sessions, e-mail, etc.

Using a standard such as OSI is a systematic way of breaking up the many tasks of a network protocol into sublayered tasks. Each layer is not required for a network simulation or even a functional system. In some cases the simulation may just be concerned with one layer such as the Data Link Layer. In this project, the most concern is in the Data Link Layer and Physical Layer. These two layers can dictate the topology of the network as well as the performance capabilities. The three MAC paradigms studied (i.e. deterministic reservation, round robin, and contention) are somewhat of an abstraction from the sonar array network models previously discussed. The reservation protocol used in the baseline unidirectional protocol disregards the FIFO request mechanism since it assumes deterministic traffic from each node. To arbitrate fairness, the round robin technique uses a specific implementation called the non-exhaustive technique. The register insertion ring and bidirectional insertion array use contention to write onto the network, although collisions cannot occur on these point-to-point topologies as in bus-based contention networks. These protocols will be studied in further detail in Chapter 4.

2.3 Beamforming Theory

Beamforming is the name given to a wide variety of sonar array processing algorithms that optimize a sonar array's gain pattern in a direction of interest. "Conventional beamforming algorithms use modern digital signal processing algorithms (such as the FFT) to great advantage, resulting in some of the most efficient array processing algorithms known" [JOHN93]. A sonar array coupled with a beamforming algorithm acts as a spatiotemporal filter that is used to separate incoming signals according to their directions of propagation and their frequency content. The characteristic parameters of a sonar array defines the array's transfer function, which is also known as its directivity pattern or aperture smoothing function. A typical directivity pattern of a linear sonar array with eight sensors is shown in Figure 2.3.1.

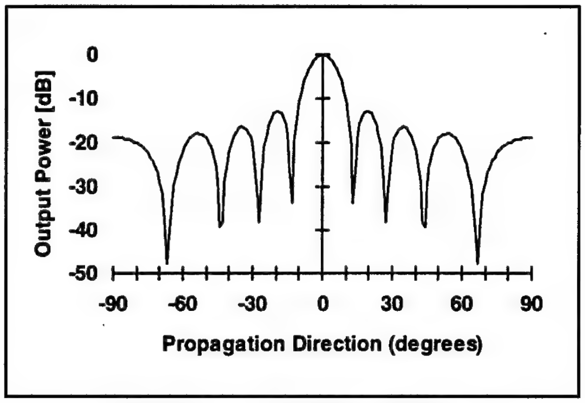


Figure 2.3.1 : Directivity Pattern

A typical directivity pattern of a linear sonar array with eight sensors, with propagation direction on the horizontal axis and output power in decibels on the vertical axis.

The directivity pattern of a particular array is analogous to a digital filter's impulse response. The highest lobe in an array's directivity pattern is called the mainlobe or a beam. The area within 3 dB of the maximum response axis (zero degrees in Figure 2.3.1) is called the mainlobe region. The mainlobe region defines the angular range of propagating signals that pass through the filter. The sidelobe regions are the angular regions where signals propagating in those directions are attenuated.

The width of the mainlobe region, which defines the angular resolution, the height of the sidelobes, and the number of sidelobes are all functions of the characteristic parameters of the sonar array. The characteristic parameters of the array include the number of sensors in the array, the distance between each sensor, the signal sampling frequency employed in each sensor, the frequency band passed by each sensor after bandpass filtering, and the weight assigned to each sensor's input. All of these array parameters must be carefully considered when designing a sonar array.

The structure of the aperture smoothing function's sidelobes varies as the number of sensors in the array is changed. Decreasing the number of sensors increases the sidelobes' amplitudes. The distance separating each sensor and the number of sensors in the array affects the angular resolution. With constant intersensor spacing, the mainlobe width decreases as the number of sensors increases. With the number of sensors fixed, as the distance between each sensor increases, the mainlobe width decreases. Further complicating array design is the problem of spatial and temporal aliasing.

The use of digital hardware requires that the input signal at each sensor be filtered to restrict the signal's bandwidth to frequencies of interest before being time-sampled. The input signal is also sampled in space by the arrays individual sensors. Both time and space sampling can introduce aliasing in a sonar array. From Nyquist's Sampling Theorem, it can be shown that spatial aliasing occurs when the input signal is not bandlimited to frequencies below $\pi c/d$, where d is the intersensor spacing and c is the speed of propagation (approximately 1.5 km/s in sea water) [JOHN93]. Restating the spatial sampling criteria, the intersensor spacing must be less than the product of π and the propagation speed divided by the maximum incoming signal frequency or $d_{\max} = \pi c / B_{\max}$. Spatial aliasing produces grating lobes in the aperture smoothing function as depicted in Figure 2.3.2.

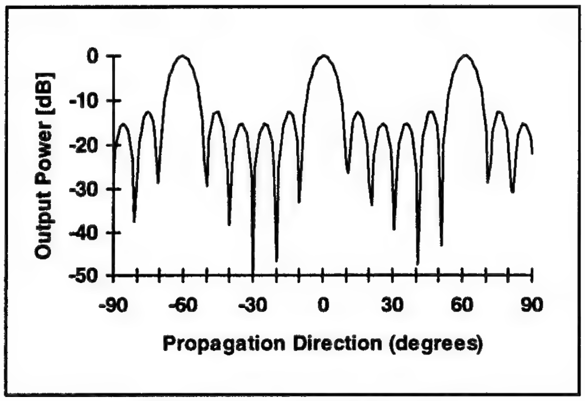


Figure 2.3.2 : Grating Lobes

Grating lobes appear in an array's aperture smoothing function due to spatial aliasing.

Grating lobes allow signals propagating in multiple directions to pass as if they were propagating in the steered direction. This phenomenon must be avoided in order to extract meaningful information from the sonar array. Assuming a signal whose highest frequency is B_{max} Hz, the maximum sampling period that can be used without causing temporal aliasing according to Nyquist's criteria is, $T_{\max} = \pi / B_{\max}$ seconds. As previously discussed, the largest sensor separation that can be used before spatial aliasing begins to occur is $d_{\max} = \pi c / B_{\max}$ meters. Thus the spatial and temporal sampling intervals are related by $d_{\max} = cT_{\max}$.

The weights applied to each sensor's input are usually defined as a window function to control mainlobe width and sidelobe magnitude [HAMP95]. For the detection of a single plane wave in isotropic noise, a uniform weighting function is approximately optimum [BURD91]. In the presence of interfering plane waves at angles outside the mainlobe or with nonisotropic noise, it is often desirable to reduce the sidelobe levels below those obtained with uniform weighting (rectangular window function). The rectangular window produces the highest sidelobe levels of any weighting function, which means the rectangular window does not attenuate signals outside the main lobe as well as other weighting functions. Several window functions including triangular, Hanning, Blackmann, Bartlett, Gaussian, and Dolph-Chebyshev are often used, but all weighting functions other than rectangular produce wider beamwidths, thus decreasing the spatial resolution. A particular window function should be chosen based on the types of signals and noise that best matches the application. The process of applying a window function to a sonar array is often referred to as array shading. The desire to reduce sidelobe amplitude, increase angular resolution, and avoid spatial aliasing forms a discrete optimization problem.

Beamforming algorithms implement a filter that passes signals propagating in a range of directions and attenuates those signals propagating in other directions. Beamforming algorithmically steers the sonar array's mainlobe beam toward desired directions. Beamforming works by delaying each sensor's output by an appropriate amount prior to summing the output of each sensor to reinforce the incoming signal with respect to noise or waves propagating in different directions. The delays that reinforce the signal are directly related to the amount of time it takes for the signal to propagate between sensors as depicted in Figure 2.3.3. A linear sonar array is depicted in the Figure 2.3.3. Sonar arrays are often configured in other geometric shapes; however, since this project specifies a linear array, the remaining theory sections pertain to a linear sonar array.

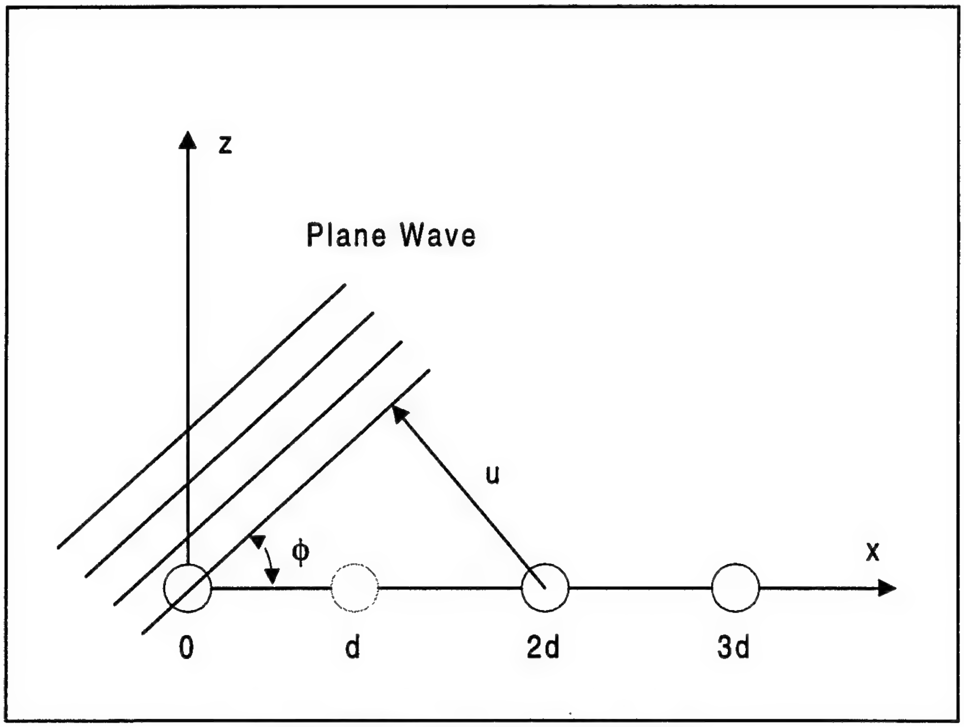


Figure 2.3.3 : Sonar Array and Wavefront

An illustration of the delay in time (or phase) of a wavefront, modeled as a plane wave, as it is seen by the different sensors in a linear sonar array.

To form a beam in the direction ϕ, each sensor signal $x_m(t)$ should be delayed by $t_m = m\frac{d}{c}\sin\phi$ seconds. Figure 2.3.4 illustrates a beam steered 45 degrees to the right using unity weighting.

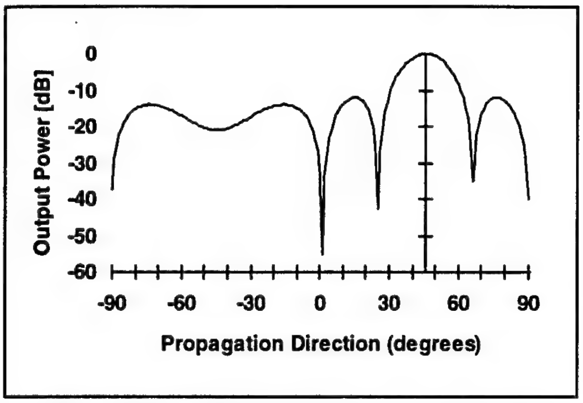


Figure 2.3.4 - Beamsteered Directivity Pattern
The figure shows the directivity pattern of an array that is steered to 45 degrees.

The average power in the beamformer output can be calculated to determine the strength of a signal in the specified direction. The direction of propagation of the strongest signals can be determined by examining the average power of the output of the beamformer as it filters at different directions. This information can then be analyzed to determine the presence of a source, the direction from which the source is propagating, and possibly its range and identity.

2.3.1 Input Signal Representation

The two general approaches used to model the input signal to a sonar array are the plane wave and the spherical wave. In each case, the signal is sampled in time and quantized to yield the signal *s(n)*. The input signal may be viewed as a plane wave when the source is in the far field, where waves approaching the sonar array have a negligible curvature. If the source is located far from the array the wavefront of the propagating wave is a plane wave and the direction of propagation is approximately equal at each sensor. The only parameter that the beamformer can detect from a plane wave is the angle from which it arrives. By considering the distance between the sensors and the speed of propagation, the signal received at each sensor can be calculated. The wave received at sensor *m* located at x_m is:

$$s(n) = \sin[\omega n + \frac{(m-PC)\omega d}{c} \sin \frac{\pi \phi}{180}] \qquad \text{(Eq. 2.3.1)}$$

where: ω = angular frequency of the incoming wave in cycles per second
n = sampling time (an integer index)
ϕ = arriving angle of incoming wavefront in degrees
PC = phase center of the array (in terms of the internode spacing d)

Since the plane wave is only indicated by its incoming angle, a source in the far field can only be located by its angle from the array.

If the source is located close to array, the wavefront of the propagating wave is curved with respect to the dimensions of the array and the wave propagation direction depends on sensor location. Using the geometry from the array in Figure 2.3.5, the input signal from spherical spreading can be derived.

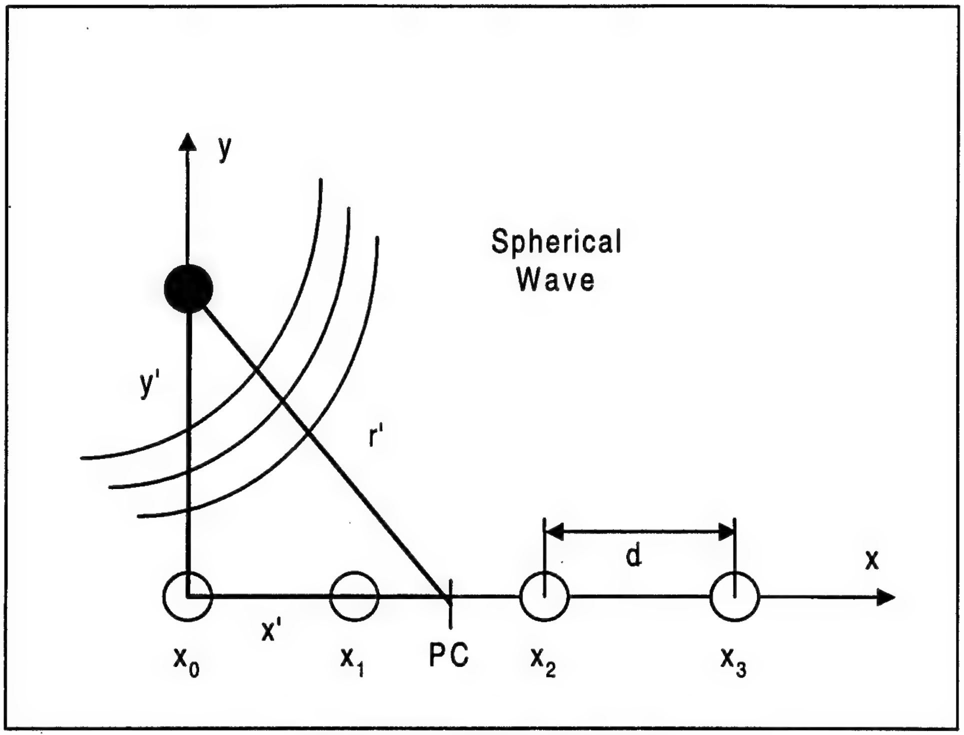


Figure 2.3.5 - Spherical Wavefront
An illustration of a wavefront modeled as a spherical wave rather than as a plane wave.

The equation is very similar to the plane wave, except that the radial direction from the phase center is specified in addition to the incoming angle. To describe a signal coming from a source located at (x', y') relative to the phase center, the radial distance from the source to the phase center is calculated by equation 2.3.2 and the radial distance from the source to sensor *m* is calculated by equation 2.3.3.

$$r' = \sqrt{(x')^2 + (y')^2} \qquad \text{(Eq. 2.3.2)}$$

$$r_m = \sqrt{(x_m)^2 + (y_m)^2} \qquad \text{(Eq. 2.3.3)}$$

Using the above equations, the signal can be described by equation set 2.3.4.

$$\begin{aligned} s_m(n) &= \sin\left(\omega n + \frac{r_m}{c}\right) \\ s_m(n) &= \sin\left(\omega n + \frac{\sqrt{{y_m}^2 + {x_m}^2}}{c}\right) \qquad \text{(Eq. 2.3.4)} \\ s_m(n) &= \sin\left[\omega n + \frac{\sqrt{{y_m}^2 + (PC + x' - dm)^2}}{c}\right] \end{aligned}$$

Each type of signal input has its disadvantages and advantages. The plane wave is a simpler wave to model, but the beamformer can only locate the incoming angle of the source. The spherical wave is more complex, but the beamformer is able to give a more accurate location of the source. By no means are the input signals limited to these two forms. In addition to the waveform itself, the source can be degraded if the signal is noisy, attenuated, or mixed with other signals.

2.3.2 Delay-and-Sum Beamforming

The delay-and-sum beamformer functions by applying a time delay Δ_m and an amplitude scaling weight w_m to the output of each of the M sensors in the array. The output $z(t)$ is then computed from the sum of the resulting signals. What does $z(t)$ show? The delay-and-sum beamformer's output is defined to be

$$z(t) = \sum_{m=0}^{M-1} w_m y_m(t - \Delta_m) \qquad \text{(Eq. 2.3.5)}$$

where $y_m(t)$ is the input measured by the mth sensor at time t.

One of the key problems in developing a delay-and-sum beamformer for sampled signals is the inability to implement arbitrary delays. Delays are restricted to multiples of the sampling period. The discrete time delay-and-sum beamformer is described by

$$z(n) = \sum_{m=0}^{M-1} w_m y_m(t - n_m) \qquad \text{(Eq. 2.3.6)}$$

where n_m is the integer delay associated with the mth sensor. The steering directions that are available to a delay-and-sum beamformer are restricted by the equation

$$\phi = \sin^{-1}\left(\frac{-qcT}{d}\right) \qquad \text{(Eq. 2.3.7)}$$

where c is the speed of propagation, T is the sampling period, d is the distance between each sensor, and q is any integer [JOHN93]. As previously discussed, the spatial and temporal sampling intervals are related by $d = cT$. Substituting this relationship into equation 2.3.7 produces

$$\phi = \sin^{-1}(-q). \qquad \text{(Eq. 2.3.8)}$$

The only possible solutions for ϕ in this equation are 0, 90, or –90 degrees because q must be an integer. This analysis reveals that sampling at the minimum acceptable rate in space and time provides only three possible delays and three corresponding steering directions when using delay-and-sum beamforming. Obviously three steering directions is less than adequate. To form more beams using integer delays, d and T must be adjusted so that d is much larger than the product of c and T. Care must be taken when designing for values of d and T. As d is increased, the risk of spatial aliasing increases; likewise, as T is reduced, the risk of temporally oversampling the input signal increases. The inability of discrete-time beamformers to delay sensor outputs exactly can be regarded as wavefront aberration [JOHN93].

2.3.3 Time-Domain Interpolation Beamforming

Interpolating between the samples of the sensor signals is often used to reduce aberrations introduced by delay quantization. Sensor signal interpolation requires that all the sensor signals be interpolated by a factor *L* to obtain oversampled signals. In an interpolation beamformer, each sensor's output is passed through an upsampler and a lowpass interpolation filter having a cutoff frequency of π / L. A beamforming algorithm then processes these higher-sampling-rate signals. The output of the beamformer is then downsampled by a factor of *L* to obtain the original sampling rate. This process is depicted in Figure 2.3.6.

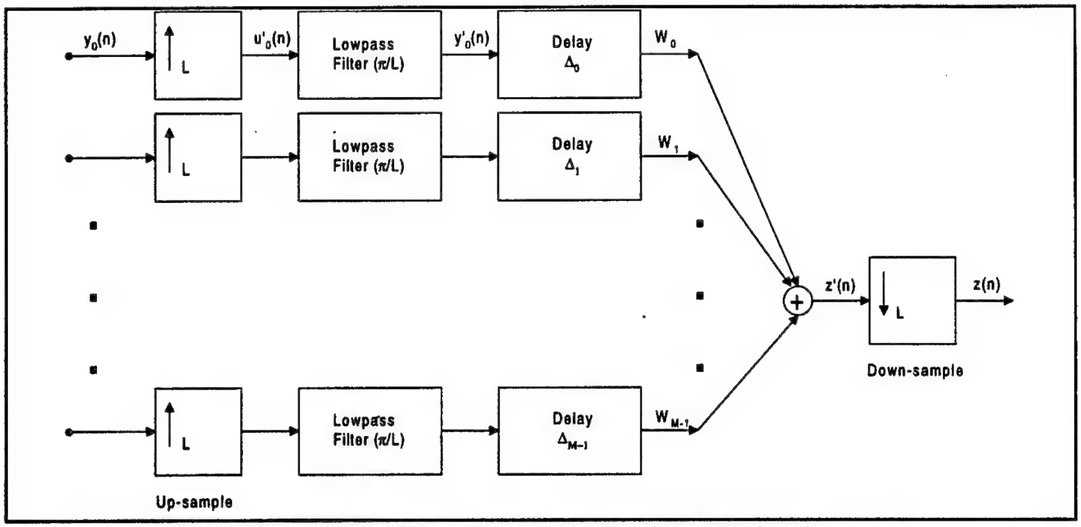


Figure 2.3.6 : Time-Domain Interpolation Beamforming

The time-domain interpolation beamforming process is illustrated schematically in the above block diagram.

A common method for producing a time-interpolated sample stream is through the use of a digital lowpass filter. A common lowpass filter is a FIR (finite impulse response) filter, which is characterized by equation 2.3.9 [IFEA93].

$$y(m) = \sum_{n=0}^{N-1} h(n)x(m-n) \qquad \text{(Eq. 2.3.9)}$$

where: y(m) = filtered data
$h(n)$ = filter coefficient
$x(m\text{-}n)$ = input signal delayed by n
N = number of coefficients

In designing the FIR filter, the objective is to find the coefficients that best fit the specifications of the filter without excessive computation requirements. The number of coefficients is determined by an acceptable trade-off between the number of calculations required and the resolution of the output for an implementation. Some techniques for finding the coefficients include the Fourier, window, and optimal methods. Since the coefficients of the filter are usually calculated off-line and are often hardwired into the filter, any coefficient method could be chosen as long as the specifications are satisfied. The Fourier method provides coefficients sufficient for interpolation purposes while minimizing the amount of overhead incurred in the delay-and-sum with interpolation algorithm. The Fourier coefficients can be found by equation 2.3.10.

$$h(n) = 2f_c \frac{\sin(n\omega_c)}{n\omega_c} \qquad \text{(Eq. 2.3.10)}$$

where: f_c = cutoff frequency

For the situation where stopband attenuation, passband ripple, and transition width are provided as specifications, a more precise method of coefficient calculation should be used. For example, the window method is simply an extension of the Fourier method in that it takes the coefficients calculated by the Fourier method and multiplies them by a windowing function that shapes the filter. The optimization method, based on equiripple passband and stopband, involves an adaptive method in which the error between the desired response and the actual response is minimized.

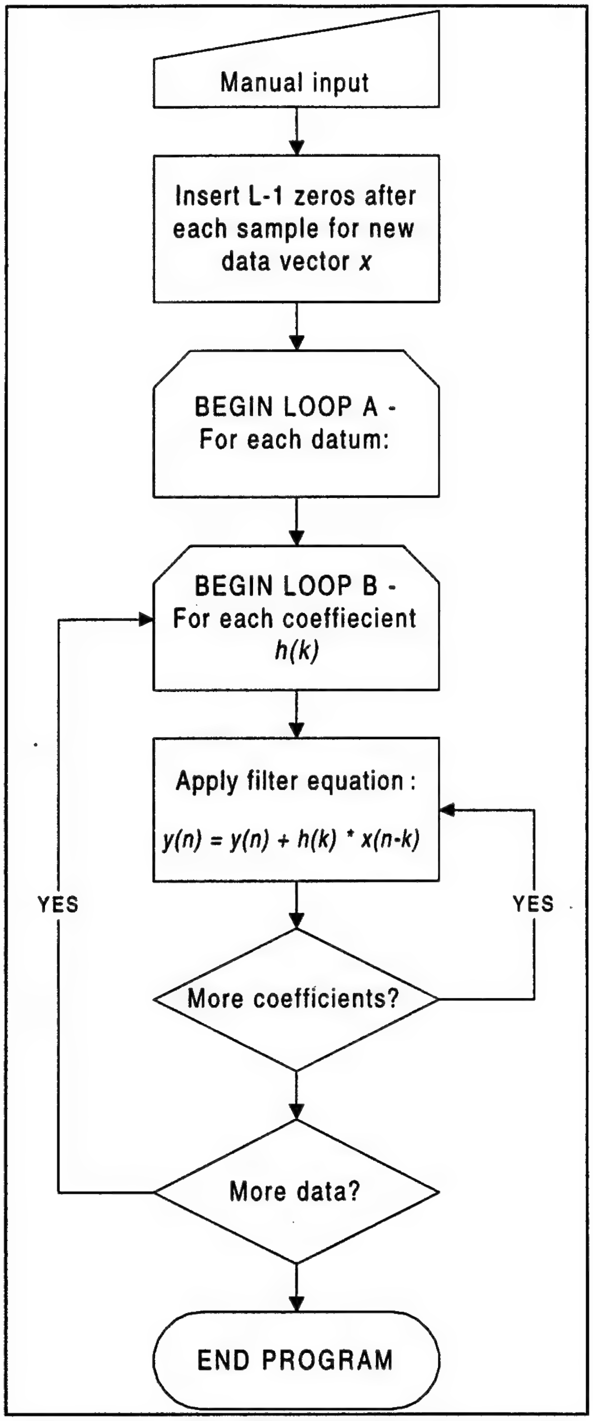

Manual input
Insert L-1 zeros after each sample for new data vector x
BEGIN LOOP A - For each datum:
BEGIN LOOP B - For each coeffiecient h(k)
Apply filter equation :
y(n) = y(n) + h(k) * x(n-k)
YES
YES
More coefficients?
More data?
END PROGRAM

Figure 2.3.7 : Interpolation Beamforming with LPF
Flow chart of the interpolation process by the use of a lowpass filter.

The conceptual algorithm of the lowpass filter is given in Figure 2.3.7. This flowchart indicates how to apply the mathematics of a FIR filter to a computer program. The advantage of using a lowpass filter to perform the interpolation rather than mathematical interpolation (e.g. spline) is that the effects of quantization and aliasing are reduced. Interpolation of the input signal also makes multirate signal processing possible. Multirate signal processing, which is the processing of data at more than one sampling rate, has been extensively researched by two scientists in particular, Ronald Crochiere and Lawrence Rabiner, in the 1970's and early 1980's [CROC81]. By interpolation, the sampling rate is mathematically increased by a factor *L*, such that the new sampling frequency appears to be $f_s' = Lf_s$. For an input signal *s*(*t*) sampled at f_s, *L*-1 new values must be inserted between each pair of samples in *s*(*n*). These new values are initially zero, and the new data stream is lowpass filtered, resulting in spreading the energy of the original samples over the whole data stream. This effectively attenuates each value by *L*.

The interpolation of the input signal via a low-pass filter allows a smoother and more resolute signal to be analyzed which may help to eliminate sampled noise. Also, because the sample spacing has been reduced by a factor of *L*, the angular resolution improves allowing a greater number of steering directions [JOHN93].

2.3.4 Autocorrelation Approach to Beamforming

An alternative time-domain beamformer is achieved by performing an autocorrelation between the data collected by the sensors. By forming a beam in the direction $\theta_b = \frac{180}{\pi}\sin^{-1}\frac{bc}{df_s}$ degrees and calculating the sum along the wavefront as shown in equation 2.3.11, the time delay beamformer can be calculated.

$$Y_n(\theta_b) = \sum_{m=0}^{M-1} s_m(n + \Delta_{m\psi}) \qquad \text{(Eq. 2.3.11)}$$

where: *s(n)* = source signal
M = total number of sensors
m = sensor number
n = sample number
$\Delta_{m\psi}$ = delay incurred at sensor *m* by source direction ψ
d = spacing between sensors
f_s = sampling frequency
c = speed of wave propagation
dir = number of steering directions
b = integer ranging from -*(dir/2 - 0.5)* to *(dir/2 - 0.5)*

The power spectrum can be calculated by forming the autocorrelation function. A crosscorrelation function quantifies the relationship between two sets of data by summing the products of the time-shifted sets. If the data have some pattern to them, then the correlation is high. If the two sets of data are noisy, then ideally the correlation is zero because all numbers are equally likely to occur in either set. In some cases, the signals may be highly correlated, but out of phase, which is why a lag is needed. For example, two sine waves 90 degrees out of phase will have a correlation of 0, but obviously the correlation should be high because they are both sine waves. By delaying one of the signals 90 degrees, the correlation will reach a maximum, and it is at that point that the two sets of data are determined to be correlated. In practice, the phase between the two sets of data is not known. A variety of lags should be investigated. The autocorrelation function is the correlation of a set of data with itself. With no lag, the autocorrelation function takes on a special property—it results in the energy of the waveform. If the autocorrelation function of the wavefront is investigated, the presence of a source can be determined by a relatively high correlation. Equation 2.3.12 gives this autocorrelation function.

$$R_{\psi}(l)=\sum_{n=0}^{N-1} Y_n(b)Y_{n+l}(b) \qquad \text{(Eq. 2.3.12)}$$

where: l = lag
N = number of data samples

By performing this autocorrelation for all incoming source angles, the beam pattern can be determined.

The autocorrelation approach beamforming algorithm can be described as the delay-and-sum algorithm with an autocorrelation operation performed afterwards for additional signal processing. The autocorrelation approach effectively is a delay-and-sum with a post Fourier transform. The main advantage of this technique is that it beamforms and calculates the power spectrum in the same series of sums. The computational load is distributed evenly over the array by a simple shift and add process in which the data is sent only once around the array.

2.3.5 Frequency-Domain Beamforming

Time-domain delay-and-sum beamforming algorithms compute their output by first delaying and then summing the sensor signals. This time-domain method of beamforming has frequency-domain counterparts. The time delay transforms to a phase shift in the frequency domain. Frequency-domain beamforming implements the beamforming calculations in the frequency domain by Fourier transforming the inputs, applying the spatiotemporal filter, and inverse transforming the result. Frequency-domain beamforming is distinctly different from time-domain delay-and-sum beamforming in that delays are not implemented by physically delaying signals before addition. Instead, frequency-domain beamforming relies on the time-shifting property of the Fourier transform to allow delays to be implemented by complex-number multiplications. If the incoming signal is $f(t)$ and its transform is $F(\omega)$, then a time-shifted signal $f(t\text{-}t_0)$ has a transform $F(\omega)e^{-j\omega t_0}$.

The FFT algorithm first takes the Fast Fourier transform of all sensors' signals. It then makes a copy of the signals in memory and multiplies these copies by the correct complex exponential for the desired angle to search. The reason a copy is made is so that the transformed signal stays in memory unaltered for subsequent loops. Also at this point, the algorithm applies a weighting window, although this can be done before the transform. The algorithm then sums the signals sample-by-sample across sensors and inverse transforms the resulting summed signal. The inverse transform has real and imaginary parts. The algorithm multiplies each sample by its conjugate, which results in a representation of the square of the filtered signal. The maximum amplitude of this signal is stored in memory. The algorithm then loops back to just after the original transform in order to multiply by the complex factor for the next search angle and repeats. At the end, the program knows the signal amplitudes for each search angle and plots them as a function of search angle. The peaks in the plot show the location of signals in the spatial-domain.

A further in-depth explanation of implementing the complex delay factors follows. For a monochromatic source, for a given sensor, the signal received is $\sin\left[\omega\left(t - D/_{c}\right)\right]$, where c is the speed of sound in water. The variable D is either the distance from that sensor to the source (which applies to a spherical wave representation) or the perpendicular distance traversed by the wave to the sensor from some reference plane (for a plane wave representation). The reference plane is a plane parallel with the wavefront, passing through the first sensor. Therefore, the phase difference in the signal received at different sensors is the result of the extra perpendicular distance Δd that the wave travels to the sensor after hitting the first sensor.

Figure 2.3.8 shows how to determine the Δd in terms of variables the beamformer knows and the incoming angle. The incoming angle, θ, is measured from the normal to the array and is positive if it arrives from the far side of the array. In the figure, since the sensors are numbered starting at the left, the right side of the figure is the far side of the array. With the signal arriving from the left of the figure, the angle θ is a negative number, so the physical representations of the angle in the figure are $-\theta$. With that explanation, the Δd for the mth sensor is $md\sin(-\theta)$, where d is the sensor spacing.

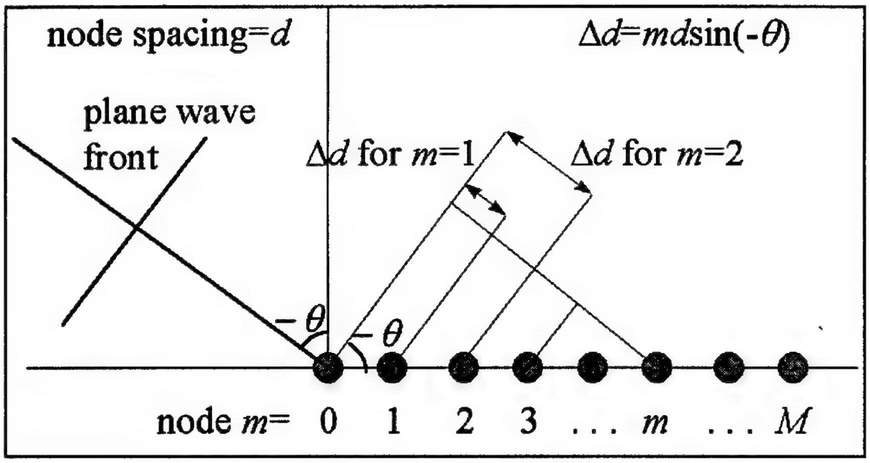


Figure 2.3.8 : Calculation of Δd

The figure shows how the calculation of Δd for each sensor is geometrically determined.

The variable ω must also be found in terms of variables the system knows. In the time-domain, the samples represent points in time starting at t=0 seconds and ending at $t = {(N-1)}/{f_s}$ seconds, where N is the number of samples taken and f_s is the sampling frequency in Hertz. When the sample set is transformed to the frequency-domain, the samples represent frequencies, each f_s/N Hertz apart, from $f = 0$ Hertz to $f = \dfrac{(N-1)f_s}{N}$ Hertz. Using an index variable n from zero to N-1 to represent each sample, the frequency is expressed as $\dfrac{nf_s}{N}$.

Finally, for the incoming signal $\sin\left[\omega\left(t - \Delta d/c\right)\right]$, the phase factor (by applying the time-shift property) is $\exp\left[-j\omega\left(\Delta d/c\right)\right]$. The delay factor by which the beamformer must multiply each sensor's signal to cancel the phase is $\exp\left[j\omega\left(\Delta d/c\right)\right]$. Plugging in the equations for ω and Δd, the delay factor for the nth sample of the mth sensor must be $\exp\left[j \cdot 2\pi \cdot \dfrac{nf_s}{N} \cdot \dfrac{md\sin(-\theta)}{c}\right]$, keeping in mind that both index variables n and m start at zero.

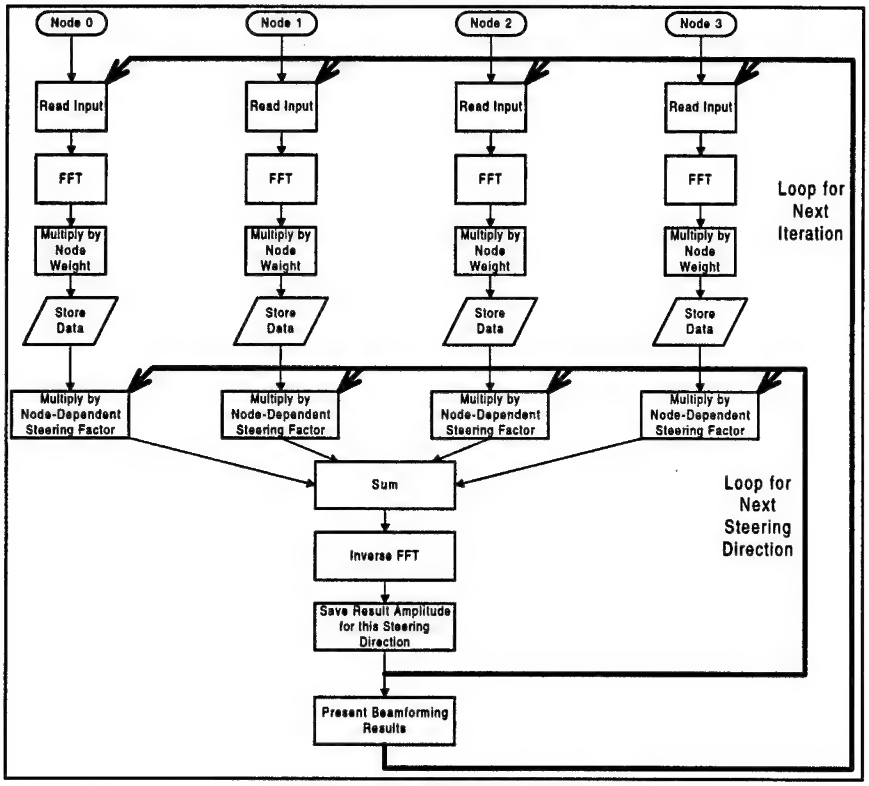

Node 0
Node 1
Node 2
Node 3
Read Input
FFT
Multiply by Node Weight
Store Data
Loop for Next Iteration
Multiply by Node-Dependent Steering Factor
Sum
Loop for Next Steering Direction
Inverse FFT
Save Result Amplitude for this Steering Direction
Present Beamforming Results

Figure 2.3.9 : FFT Flow Chart

The flow chart depicts the FFT beamforming algorithm, where the dark lines indicate iteration loops.

Although frequency-domain beamforming introduces more complexity, it offers several notable advantages. One key advantage of frequency-domain beamforming is that the beams can be steered to any direction, unlike the time-domain beamformers where the number of steering directions is limited due to the restriction of integer delays. Another advantage is that the sampling frequency does not effect the beam steering resolution [HAMP95]. The ability to compute the Discrete Fourier Transform with the very efficient FFT algorithm is also an advantage of frequency-domain beamforming.

2.3.6 Parameter Estimation

Parameter estimation techniques are often used as an alternative to conventional beamforming. Rather than performing spatiotemporal filtering, parameter estimation techniques attempt to estimate the amplitude and direction of propagation of each wave received by the sonar array. The estimation is modeled as an optimization problem to find the parameter values that minimize some error function. The parameter estimation techniques listed in the Beamforming Taxonomy in Chapter 9 were reviewed but were not considered for immediate implementation. Conventional beamforming has been the primary focus so that a fundamental understanding sonar signal processing could be developed. Also, the system that the algorithms will be implemented on is a real-time system where, historically, beamforming techniques have been while parameter estimation has been used for off-line signal analysis [JOHN93]. Parameter estimation techniques will be studied in further detail with the hope that the parallelization of some of these techniques will provide enough added efficiency which will make them realizable on a distributed real-time sonar array.

2.4 Parallel and Distributed Computing

Parallel and Distributed Computing (PDC) is a very broad and deep field of study, and it is important to gain a basic understanding in the background of PDC before examining the approach taken in this research to simulate the sonar array problem. The information covered in this chapter includes parallel architecture, interconnection networks, and processor communication. Where applicable, how the sonar array relates to the scheme of parallel and distributed computing is addressed. The architecture of parallel computers has been divided into sets of paradigms and classes which over time have become the de-facto standards in classifying the general architecture of a parallel computer.

2.4.1 Paradigms

In order to simplify the field of computing, architectures are divided into paradigms based on the number of data and instruction streams that are handled at one time. One such set of paradigms was proposed by Michael Flynn in 1966, which was the same time period that the first parallel computers were introduced. Flynn's taxonomy consists of four paradigms that classify all computers: SISD, SIMD, MISD, and MIMD [FLYN72].

1. *SISD:* Single-instruction, single-data stream computers are the conventional von Neumann architectures that decode one instruction at a time. A SISD computer has a single control unit, but may have multiple functional units to incorporate pipelining.
2. *SIMD:* By operating in a lockstep fashion, single-instruction, multiple-data stream computers perform the same operation on different pieces of data. Architectures fitting this paradigm contain a single control unit which distributes an instruction stream to multiple processors, each of which then executes the instruction on separate streams of data. A variation of the SIMD approach is SPMD, in which a single program works on multiple pieces of data. SPMD implies the execution of the processing elements is not strictly lockstep.
3. *MISD*: The least common of the four, multiple-instruction, single-data stream computers contain processors that have their own control unit but share a common data stream.
4. *MIMD:* The last paradigm, multiple-instruction, multiple-data streams, is a general-purpose parallel architecture that consists of *N* instruction streams, *N* processors, and *N* data streams [ZOMA95]. Most parallel processing machines fall into this category.

2.4.2 Parallel Architecture Classes

In addition to architecture paradigms that classify all computers—sequential and parallel—there are two major classes of parallel computers: multiprocessors and multicomputers. The distinguishing factor between the two classes is the memory architecture of the system, which dictates the communication method applied. Upon examination of these two classes, the differences between parallel computing and distributed computing become more apparent.

Multiprocessors. Often referred to as tightly-coupled machines due to the memory configuration, multiprocessors are parallel computers with multiple processors sharing direct access to memory. The memory does not necessarily have to be a single physical entity; it can exist as several memory components that are directly accessible by all processors. Nonetheless, multiprocessors generally share a single address space for all the memory devices in order to give the appearance of a single memory module. Processing by way of multiprocessors is sometimes termed "true" parallel processing because multiple tightly-coupled processors work together on the same computational problem.

Three general multiprocessor models based on memory access have been identified: UMA, NUMA, and COMA. All of the processors in the UMA (uniform memory access) model share a physical global memory that can be accessed by every processor in the same amount of time, which is how the UMA name was derived. Furthermore, a symmetric multiprocessor (SMP) is an UMA machine in which the all processors have equal access to the peripheral devices as well as memory [HWAN93]. The symmetric multiprocessor configuration is one level of architecture used for simulation of the sonar array in this research. Each SPARCstation-20/85 workstation contains two processors in an SMP configuration.

The NUMA (non-uniform memory access) model consists of processors that each have their own memory component. The individual memory components may be accessed by any other processor. Because of the locality differences of the distributed memory modules, the access time to different parts of the shared memory varies for each processor, thus creating at least two different access times: a local access time and a remote access time. In some cases, a physically-shared global memory provides a third access time that falls between the local access and the remote access times.

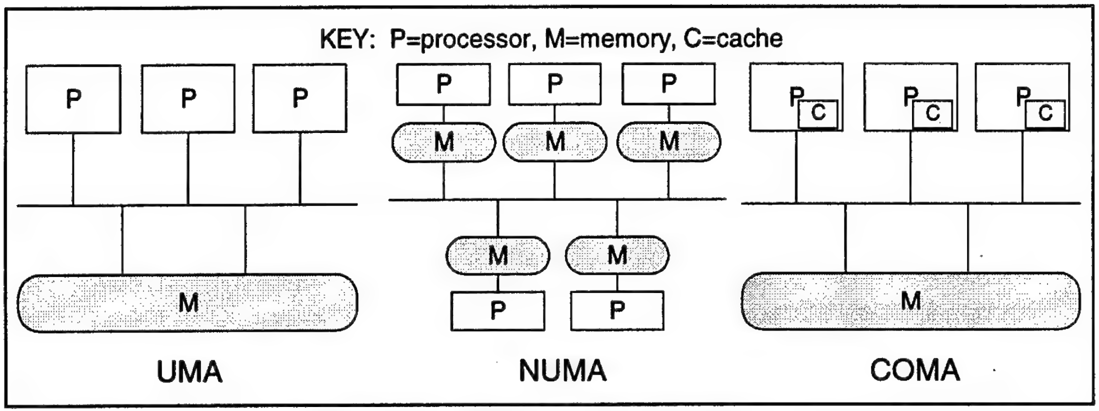


Figure 2.4.1 : Multiprocessor Configurations

This figure shows the various multiprocessor configurations (UMA, NUMA, and COMA) and depicts how the different models use global memory and distributed memory.

COMA, or cache-only memory access, computers have only local caches and global memory. A variant of the COMA model is the CC-NUMA, or cache-coherent non-uniform memory access, which has distributed memory and distributed cache directories.

The advantages of multiprocessor systems include ease of programming and the potential for excellent performance of fine-grain programs due to low communication costs. Since the memory is accessed as a single address space, the programmer does not have to be concerned with how the processors access data when needed. In addition, multiprocessors usually employ specialized processors and interconnection networks that speed up data access. However, the main disadvantage of multiprocessor systems is that, unlike multicomputers, they generally have limited scalability.

Multicomputers. Multicomputers are loosely-coupled machines which incorporate processors with their own local memories that are inaccessible by any other processor. A processor and its memory essentially form a stand-alone computer. Thus, parallel machines of this nature are classified as multicomputers. To stay consistent with the multiprocessor models, multicomputers fall under the NORMA, or no remote memory access, model. Because the memory modules are not accessible by remote processors, the processors must communicate through message passing. The array of processors proposed in the linear parallel sonar array is a multicomputer system. Multicomputers are desirable in a multitude of situations because they provide flexibility, scalability, and cost-effectiveness.

Often, computing by way of multicomputers is referred to as distributed processing instead of parallel processing. However, advances in architecture design and capabilities have reduced the distinction between parallel and distributed processing. For example, the simulation testbed for the sonar array involves both classifications. First, an upper level of parallelism exists in the workstations (or multicomputers) connected via Ethernet, SCI, or ATM. Secondly, each workstation is equipped with dual processors that provide a lower-level symmetric multiprocessor architecture, as mentioned in the multiprocessor discussion. Many modern systems carry characteristics of both multiprocessors and multicomputers, which can make it difficult to classify the parallel system into just one category. The interconnection network often plays a more critical role in determining the coupling of the system than does the memory configuration alone.

2.4.3 Interconnection Networks

The interconnection network (ICN) affects factors of the parallel computer such as the communication latency, scalability, efficient program granularity, fault tolerance, and bandwidth [ZOMA96]. The three ICNs used in the *HCS Research Laboratory* for simulation of the sonar array include bus-based Ethernet, point-to-point-based SCI, and switch-based ATM. The base data rates of the three networks range from 10-Mbps to 1-Gbps, which provides a spectrum for investigating the effects of communication in the sonar array.

Ethernet

As the most widely used local-area network today, Ethernet is a logical testbed network for determining performance of the parallel sonar array. The sonar array is not likely to have a fast interconnect because it must have low power and be disposable. The Ethernet specification was first published in 1980 and was later adopted under the IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) standard [SPUR95]. A 100-Mbps version of Ethernet called Fast Ethernet is also commercially available, and a 1-Gbps version is currently under development.

Many different types of media can be used for an Ethernet implementation as summarized below:

- Thick Ethernet: 10BASE5, coaxial cable of approximately 0.4 inch diameter
- Thin Ethernet: 10BASE2, coaxial cable of approximately 3/16th inch diameter
- Twisted-pair Ethernet: 10BASE-T, twisted pair telephone wire
- Fiber Optic Ethernet: FOIRL (Inter-Repeater Link) and 10BASE-F.

In acronyms such as 10BASE5, the 10 stands for the bandwidth in Mbps, BASE stands for baseband which is the signaling method, and the 5 indicates the maximum segment length in meters when multiplied by 100. The *HCS Research Laboratory* uses both 10BASE2 and 10BASE-T. The simulations of the sonar array are run using both of the 10BASE connections on a cluster of eight SPARCstation-20/85 machines, each with dual CPUs in an SMP configuration.

Scalable Coherent Interface (SCI)

The Scalable Coherent Interface is a standard describing an interconnect that provides cache-coherency and communication at speeds of up to 1 gigabyte per second per link between processors [SCI92]. The interconnect models a Processor-Memory-I/O bus by indicating the data, source, and destination in a single address space [BECK96]. In other words, it is sufficient to use Load and Store commands to access data on any part of the SCI network; the underlying transport is transparent to the user. Unlike other networks, SCI uses a distributed cache coherence mechanism to keep caches up to date and to hide latency. If a processor changes the data, all cached copies of the data are notified and declared stale. If necessary, fresh copies replace the stale copies in the caches.

The SCI testbed in the *HCS Research Laboratory* is configured into a switched ring of eight dual-CPU SPARCstation-20/85 workstations using Dolphin SCI/Sbus-1 adapter cards, electrical distribution units (EDUs) that carry both the incoming and outgoing signals, and cable sets, with peak bandwidth at 1-Gbps per link, but it does not yet support cache coherency. This SCI configuration provides one of the networks over which the beamforming programs are run.

Asynchronous Transfer Mode (ATM)

The third network, Asynchronous Transfer Mode, is a networking technology that can be used for transmitting data, video, and voice over local area networks (LANs), metropolitan area networks (MANs), and wide area networks (WANs) [FORE96]. ATM is the implementation of broadband ISDN. It is a promising technology that promises to unite the telecommunications and computer industries. Although ATM is much different than other high-speed computer networks, it can be leveraged as a multicomputer interconnect. The ATM testbed in the *HCS Research Laboratory* is capable of 155-Mbps per link and is arranged in a star topology using the same eight dual-CPU SPARCstation-20/85 workstations as the SCI switched ring and Ethernet bus.

2.4.4 Communication and Programming

The computer architecture and programming model can relate in two ways: either the programming model resembles the inherent communication dictated by the architecture or it hides the architecture by making it transparent to the user. For example, multicomputers are physically arranged for a message-passing environment. With certain parallel processing languages, multicomputers can actually mimic a shared-memory environment at the programmer level. One such language that approaches parallel programming in this way is Linda [CARR90]. The distinction between logically-shared memory and physically-shared memory is transparent to the user. Likewise, shared-memory machines can mimic message passing at the programming level. Each approach has its disadvantages and advantages depending on the desires of the programmer. Programming in a shared-memory environment is easier for the programmer because he does not have to be concerned with how to get data from one processor to the next.

On the other hand, programs based on message passing scale more smoothly. In any case, the model greatly affects the effort required to design a parallel program.

There have been a large number of attempts to create the perfect environment in which a parallel program is created and executed. The three basic approaches are the implicit, explicit, and new-program approaches. In an ideal environment, the programmer simply feeds the compiler a sequential program from which it generates the parallel program implicitly. The compiler determines all of the inherent algorithmic parallelism and communication necessary to execute the program in parallel. This implicit approach has been implemented in a number of languages, but it is only remotely as efficient as explicit parallelism. In languages using the explicit approach, the programmer sets off the instructions to be executed in parallel by adding parallel constructs to existing languages such as C or FORTRAN. Figure 2.4.2 summarizes the explicit vs. implicit approach to parallel programming. A third technique involves using an entirely new language designed specifically for parallel programming. Although this approach often shows great efficiency, the disadvantages of using the new language are the learning curve and the potential for limited portability.

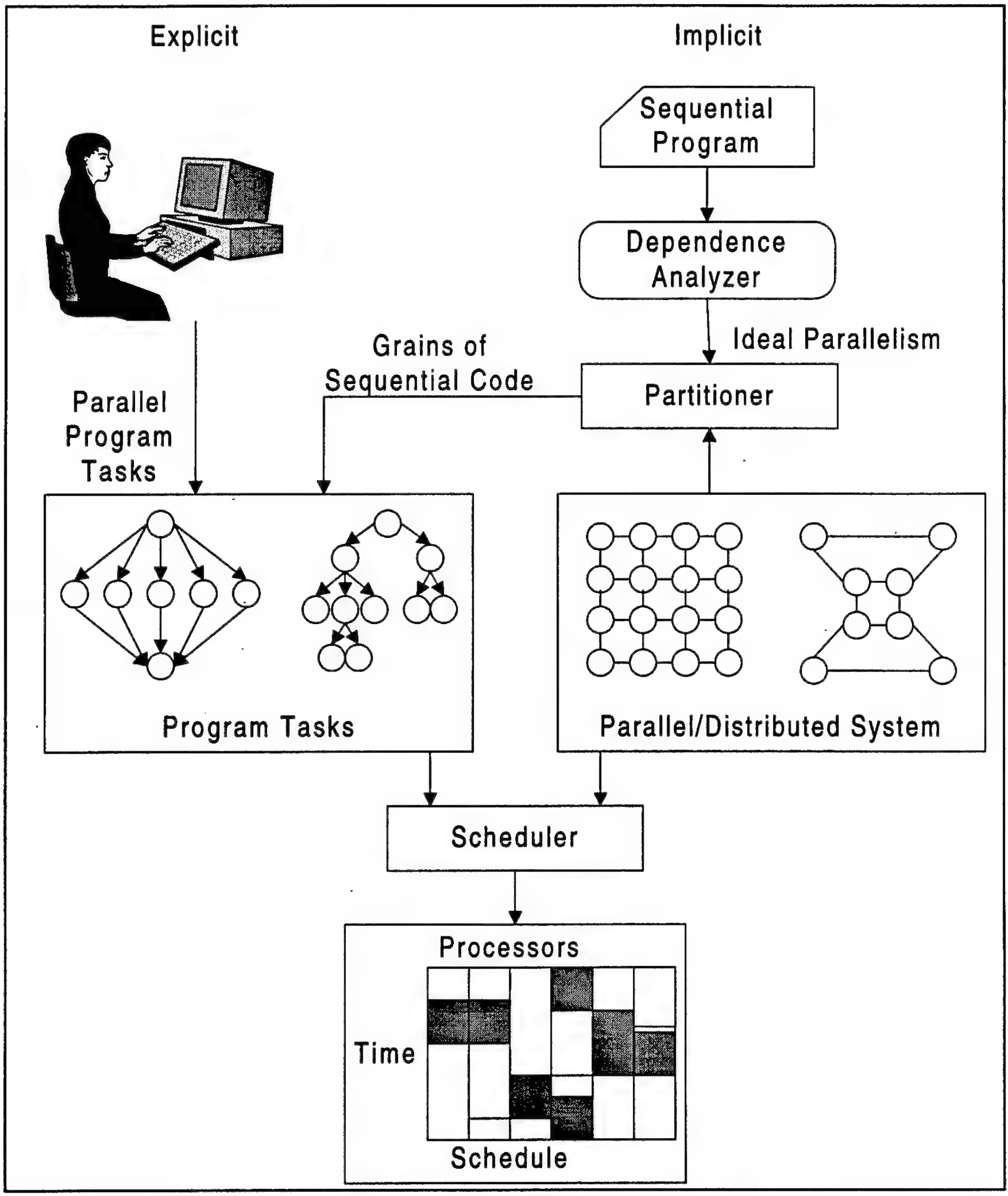


Figure 2.4.2 : Implicit vs. Explicit Parallel Programming [ZOMA96]

The left side of the figure shows explicit parallel programming where the programmer must dictate the parallelism. The right side shows implicit parallelism where the computer converts regular sequential code into parallel code.

MPI

The programming model used in this research is message passing on a cluster of workstations by way of the Message-Passing Interface (MPI), which is an adopted standard that defines the syntax of a core set of parallel communication functions [SNIR96]. This explicit approach to deriving parallel programs, such as constructing parallel beamforming programs, allows the programmer to determine exactly where and how much communication will take place by adding parallel constructs to C- or FORTRAN-based programs. This capability is important in the design of new parallel programs because it gives the user the flexibility he may require in order to discover the most efficient program. In addition, the sonar array is likely to be distributed in nature, and thus benefit from a message-passing environment.

Born out of the desire for a standardized message-passing system, MPI was created by over 80 researchers in both industry and academia, collectively called the Message-Passing Interface Forum. Some general features and capabilities of MPI include [SNIR96]:

- Is based on the needs of the application programmers
- Allows efficient communication that avoids memory-to-memory copying
- Allows overlapping of computation and communication
- Allows heterogeneous implementations
- Allows both FORTRAN and C bindings while keeping the standard language independent
- Provides a reliable communication interface
- Defines an interface similar to other current message-passing systems such as Parallel Virtual Machine (PVM) but allows greater flexibility
- Defines a portable interface that may be used on a wide variety of platforms with no significant changes to the underlying communication and software
- Allows for thread-safety by requiring that a blocking call executed by one thread does not block all threads on the processor.

With these goals set forth, the MPI Forum developed version 1 of MPI in a little over two years and released it in June, 1994. In another year, a revised version 1.1, which is the version used in this project, was released reflecting clarifications and corrections to version 1. Currently, version 2 of MPI is under development.

A large number of implementations of the standard specification have been created for a variety of configurations and platforms, such as Networks of Workstations (NOWs), the PowerChallenge, and the IBM SP2. One implementation used by this laboratory is MPICH, which was created at the Argonne National Laboratory [GROP96]. MPICH combines portability with high-performance goals, which is actually reflected in its name. The CH stands for Chameleon, indicating its environment adaptability and its quickness, in addition to the fact that the programmers derived part of MPICH from the Chameleon message-passing package [GROP]. The second implementation used for this research is MPI-SCI, which was designed specifically for Dolphin's Sbus-to-SCI cards at Parallab at the University of Bergen [PROG95].

Features of MPI. As the fundamental feature of MPI, point-to-point communication is supported in a variety of forms. The two primitives to achieve this communication are *MPI_SEND* and *MPI_RECV.*

Attached to each message-passing command is the size and location of the data and a control envelope that specifies the destination, tag, and communication group. Table 2.4.1 provides details on the syntax of *MPI_SEND* and *MPI_RECV*.

MPI_SEND(buf, count, type, dest, tag, comm)		MPI_RECV(buf, count, type, source, tag, comm, stat)	
buf	address of send buffer	buf	address of receive buffer
count	number of elements to send	count	number of elements to receive
type	datatype of send buffer elements	type	datatype of receive buffer elements
dest	process id of source process	source	process id of source process
tag	message tag (identifier)	tag	message tag (identifier)
comm	communicator	comm	communicator
		stat	status (error messages)

Table 2.4.1 : MPI Primitives Syntax

The table shows the function prototypes for MPI_Send and MPI_Recv.

MPI_SEND and *MPI_RECV* are blocking calls. Blocking indicates that the buffers used to implement the call are not available for re-use until the call is complete. MPI defines five types of calls, one of which is blocking. In a non-blocking call, the buffer may be used before the call completes, but it is up to the user to ensure that the data is not written over before the call completes. The third and fourth types of calls form the dichotomy of local and non-local calls. In a local call, the procedure is called and completed within the single process. In contrast, non-local calls require execution of a procedure on a different process. Finally, the fifth call is collective, in which all processes in a process group need to call the same procedure. In addition to different types of calls, MPI uses four modes of communication summarized in Table 2.4.2 [SNIR96].

Mode	Description
Buffered	Buffer space is provided for the send so that the send can complete before a matching receive has been posted by another process. This is allowed locally or non-locally.
Standard	Lets MPI determine whether the data will be buffered. In a send command, for example, if MPI chooses the data not to be buffered, then the send must wait for a matching receive to be posted before it starts
Synchronous	The send can be started regardless of a matching receive, but it cannot complete until the receive has been posted and has begun receiving the data.
Ready	The send is started only after the matching receive is posted.

Table 2.4.2 : Four Modes of MPI Communication

The descriptions used here are for the different types of send operations.

Some variations of the send and receive primitives that include these various modes of communication and types of calls are listed below.

- *MPI_SENDRECV*: performs both a blocking send and a blocking receive with each operation having their own control envelope within the main envelope.
- *MPI_SENDRECV_REPLACE*: performs blocking send and receive using the same buffer space, with the message sent replaced by the message received.
- *MPI_ISEND* & *MPI_IRECV*: non-blocking send and receive.
- *MPI_SCATTER*, *MPI_GATHER*: collective communication commands that allow the scattering of a vector of data from one process to all processes in a group and allow the gathering of pieces of data from processes into a vector of data in a single process, respectively.
- *MPI_BCAST*: collective communication that sends data to all processes in a group.

Additional mechanisms included in MPI provide environmental inquiry, basic timing, and a profiling interface. The MPI standard has over 100 constructs plus any that are added by individual implementations.

2.4.5 Performance Metrics

In order to measure the effectiveness of the parallelization of a program, certain concrete performance metrics must be introduced and examined. Although the overall quality of the program is both objective and subjective, the following terms are ways of objectively measuring the performance of a parallel program.

Parallelism Profile Estimate

A parallelism profile is a plot of the degree of parallelism, or number of processors in use, versus time. The degree of parallelism may or may not be bounded by the number of processors available. For example, if the program is to be evaluated based only on the amount of parallelism possible, then the number of processors available is irrelevant. On the other hand, if the actual execution of the program is to be examined, then the number of processors is a major factor in the actualized degree of parallelism.

```
m ←number of elements
                    i ← processor rank
repeat
        m ← ( m / 2 )
        if ( i < m)
                read A[ i ]
                read A[ i + 1 ]
                compute A[ i ] + A[ i + m ]
                store in A[ i ]
        end
                until m = 1
```

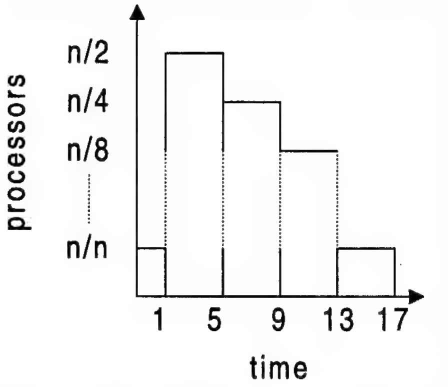


Figure 2.4.3 : Parallel Algorithm for O(log *n*) Summation and the Parallelism Profile

The left side of this figure gives the algorithm for summing the values in an array. The right side gives the degree of parallelism versus discrete time.

An illustration of a parallel algorithm and its parallelism profile for summing numbers in O(log *n*) time is given in Figure 2.4.3. The variable *m* initially represents number of elements in the array, *i* is the processor rank, and *A* is the vector of data elements. This particular algorithm assumes unlimited number of processors. The statements inside the *repeat* loop must be executed sequentially, but the iterations of the *repeat* loop may be executed in parallel. For example, for a system with 8 processors labeled 0-7, time step 0 is spent sending out the data to the processors. Time steps 1-4 involve *m*/2 processors, or 4 processors. The algorithm continues in this fashion until a single processor is left holding the final sum.

Execution Time

The execution time is defined as the time elapsed from the beginning of the program execution to the end of program execution and is an important factor in determining the responsiveness of the system. If an algorithm is to be executed in real-time, as will the processing on the sonar array, then the execution time is crucial to the success of the algorithm. The three main components of execution time include computation time, communication time, and idle time. *Upshot*, the profiling tool used in MPICH, illustrates the times spent in each of these graphically [SNIR96].

Speedup

Speedup measures the quality of an algorithm with regard to the reduction of processing time. Speedup is defined as [ZOMA96]:

$$S = \frac{T_s}{T_p} \qquad \text{(Eq. 2.4.1)}$$

where: T_s = Running time of best available sequential algorithm

T_p = Running time of the parallel algorithm

The larger the speedup, the better the quality of the algorithm. Two rules that are commonly referred to in parallel computing are Amdahl's Law and Gustafson's Law. Amdahl's Law limits the speedup to the amount of inherent parallelism found in the algorithm [AMDA67]. Amdahl's Law states the following:

$$S_N = \frac{T_S}{T_P} = \frac{T_S}{\alpha T_S + \frac{(1-\alpha)T_S}{N}} \qquad \text{(Eq. 2.4.2)}$$

where: S_N = speedup for an N processor system

α = fraction of algorithm that is not parallelizable (Amdahl's fraction)

The alternative to Amdahl's Law is Gustafson's Law. Gene Gustafson recognized that Amdahl's Law is valid only for a fixed workload size, and is thus not scalable [GUST88]. For the case where the problem size scales with the number of processors (i.e. where α is a function of the problem size), Gustafson derived speedup based on a fixed time concept that led to a scaled speedup model [HWAN93]:

$$S'_N = \frac{N}{1+(N-1)\alpha(N)} \qquad \text{(Eq. 2.4.3)}$$

where: S'_N = scaled speedup for an N processor system

$\alpha(N)$= fraction of algorithm that is not parallelizable

In many cases, speedup is the ultimate goal when parallelizing a sequential program. However, in the case of the sonar array, speedup is only one of the many reasons why the beamforming algorithms are partitioned, decomposed, and mapped into parallel forms.

Efficiency

The efficiency is defined as the speedup divided by the number of processors used. The efficiency is highest when all processors are used throughout the entire execution of the program, and lowest efficiency occurs when the program runs almost exclusively on a single processor or the communication overhead dwarfs the computation time of each process.

Other Criteria

In addition to the criteria listed above, the performance of the application is very much dependent on the feasibility of building such a device: the cost to build, power consumption, ease of programming, responsiveness, and fault tolerance. All of these metrics are extremely important to the sonar array application.

Although this chapter only briefly touches on aspects of parallel computing, enough information is provided so that techniques used to parallelize the sonar array can be understood and its niche identified in the grand scheme of parallel and distributed computing. The proposed sonar array must be a multicomputer with distributed processing for several reasons. First of all, the array will not support communication links fast enough to merit shared memory. Secondly, the very nature of the array will require that each hydrophone node have its own processor and be able to perform calculations on the data independently of what is happening on the other nodes. In order to reach data at other nodes, the information must be explicitly passed to the next node, which is a message-passing communication technique. Thus, simulating the sonar array on multicomputers constructed with clusters of workstations connected by Ethernet, ATM, or SCI represents a useful simulation environment for the sonar array with low-, medium-, and high-speed communication.

2.5 Decomposition and Parallel Programming Design

Designing a parallel algorithm is quite different from designing its sequential counterpart. In fact, some consider parallel program design to be an art form. All too often, the approach to building a parallel program is as follows: a programmer states the problem and deduces the sequential algorithm. Then some parallel solution that appears from first glance to be the best solution is sketched, analyzed, and illustrated in detail, and then implemented. There are two major drawbacks with this at-a-glance technique. First, what if the derived solution fails? With the algorithm derivation described above, the programmer finds himself starting over at the beginning from the sequential program. On the other hand, if the program is successful, how does one go about ensuring that it is the "best" solution? Answers to these questions are found in the development of a set of guidelines by which parallel programs are formulated.

Unlike sequential processes that strictly follow the von Neumann model, parallel processes do not have a universally accepted method of program development. However, certain trends in parallel algorithms can be identified as decomposition methods and programming models to serve as guides along the path of creating the parallel algorithm. In addition, these methods and models provide a way for evaluating alternatives to a chosen parallel algorithm. Many different researchers have devised their own set of models and decomposition techniques. Because of the variety of methods in use and variations in terminology, one cannot strictly look at decomposition methods alone or programming models alone to deduce a set of guidelines by which the parallel program is developed. The entire design process must be considered. The goal of this section is to develop and present a taxonomy of a variety of different approaches introduced by researchers, and then deduce a representative set that will be used in subsequent chapters as the basis for the parallel partitioning, decomposition, and mapping of two time-domain beamforming algorithms.

In building the parallel program, the three general steps taken are: 1) partitioning; 2) mapping; and 3) tuning. Defined as the procedure of breaking up the sequential program into "a set of parallel processes and data," decomposition is the crucial first step that guides the rest of program design [RAGS91]. The following subsections provide a representative taxonomy of decomposition methods.

2.5.1 Programming Paradigms and Methods

Nicholas Carriero and David Gelernter introduce a popular model by which parallel programs are designed [CARR90]. The model is divided into three paradigms and three methods, where the paradigms can be considered as the decomposition model and the methods as the decomposition strategy. The three methods are applied to each of the paradigms. The paradigms are illustrated in Figure 2.5.1.

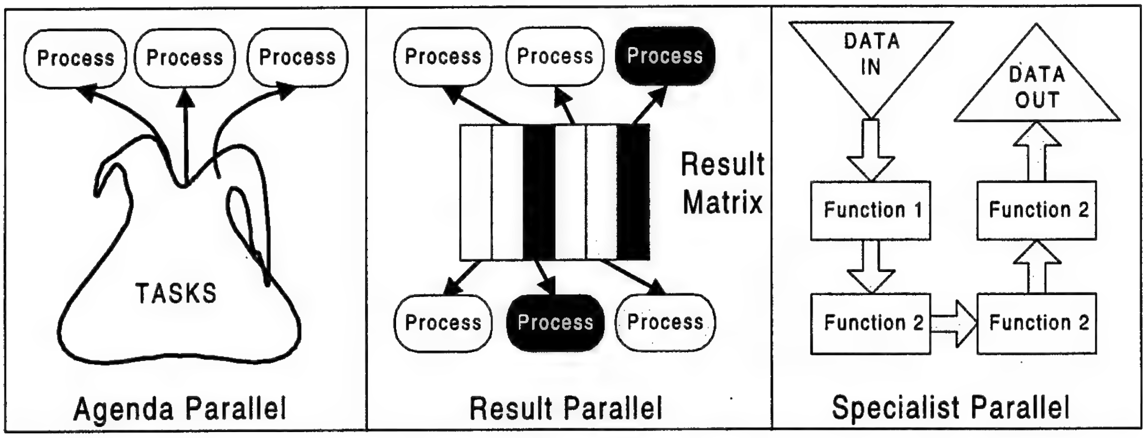


Figure 2.5.1 : Parallel Paradigms

The three paradigms for parallel programming as described by Carriero and Gelernter are agenda parallelism, result parallelism, and specialist parallelism.

The paradigms are result, agenda, and specialist parallelism. The first of these paradigms, result parallelism, focuses on the final solution expected from the algorithm. The algorithm is divided into a number of processes that each forms a piece of the result. For example, the result parallel decomposition can be applied to a matrix multiply. The final product of a matrix multiply of matrices A of size m x n and B of size n x d is an m x d matrix C. For result parallelism, each process calculates an element of the C matrix. Thus, each process forms a piece of the result.

Agenda parallelism, the second paradigm, models a set of tasks, where the tasks are processes necessary to accomplish the goal. Often this takes on a master-worker paradigm in which one process creates the list of tasks and delegates them to processes requesting work. An agenda parallel matrix multiply divides the rows of matrix A and the columns of matrix B, and creates tasks based on the multiplication and addition of these.

Finally, in specialist parallelism, each process specializes in a particular activity or function independent of the other processes. The specialist paradigm is most useful on programs with a variety of tasks, unlike a matrix multiply. In fact, a process could specialize in performing a matrix multiplication while other processes concentrate on other computations to be performed on the data. This kind of parallelism is found in many applications, such as image processing.

The three methods introduced by Carriero and Gelernter present ways to apply the aforementioned paradigms and are defined by the distribution of the data structure. The first method, message passing, encapsulates data in its own process so that no other process may access that data directly. The second strategy creates a live data structure, in which a process is associated with each element of the structure. When the process is completed, it contributes its results to the data structure and the process terminates. Each element is available only after the process containing it completes. Finally, the distributed data structure shares its data among processes. The processes communicate by leaving the data in the shared distributed data structures. These methods are illustrated in Figure 2.5.2.

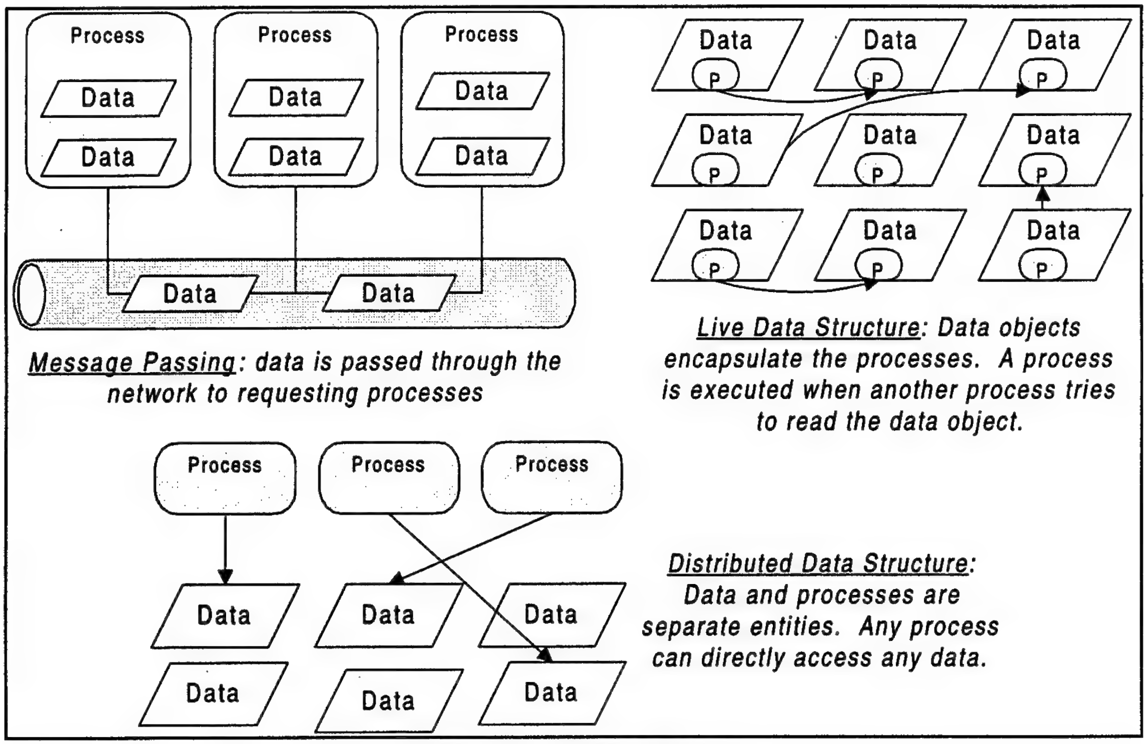

Process
Data
Data
Process
Data
Data
Process
Data
Data
Data
Data
Message Passing: data is passed through the network to requesting processes
Data
P
Live Data Structure: Data objects encapsulate the processes. A process is executed when another process tries to read the data object.
Process
Process
Process
Data
Distributed Data Structure: Data and processes are separate entities. Any process can directly access any data.

Figure 2.5.2 : Parallel Methods

The figure above illustrates the 3 methods, as given by Carriero and Gelernter, in which processes can communicate between each other: message passing, live data structures, and distributed data structures.

Through close inspection, the natural fit between the paradigms and the methods is apparent. Result parallelism fits naturally with a live data structure, agenda parallelism fits with the distributed data structure, and specialist parallelism fits naturally with message passing. However, the natural fit of a paradigm to a technique does not necessarily produce the most efficient program. Carriero and Gelernter suggest the following technique for developing a parallel algorithm [CARR90]:

1. *choose the paradigm that is most natural for the problem,*
2. *write a program using the method that is most natural for that paradigm, and*
3. *if the resulting program is not acceptably efficient, transform it methodically into a more efficient version by switching from a more-natural method to a more-efficient one.*

In order to decide which method is the most efficient, one must look at how each of the paradigms can be applied to the algorithm.

Domain and Control Decomposition

One widely recognized set of decomposition techniques includes domain decomposition, control decomposition, and perfectly parallel [RAGS91, HWAN93, LEWI92]. In domain decomposition, the data structure is partitioned among a number of processes that execute similar computations on each data set. Three types of problems suit this particular decomposition method [RAGS91]:

1. *Problems with static data structures*
2. *Problems with dynamic data structures but the structure is connected to a single entity.*
3. *Problems with a fixed-size domain but the amount of computation on each region of the domain is dynamic.*

Domain decomposition is perhaps one of the most intuitive ways to decompose a program.

When data structures are likely to be irregular, control decomposition becomes the more natural solution. Control decomposition, the alternative to domain decomposition, is based on the flow of control of the program. The two types of control decomposition are functional and master-worker. Functional decomposition distributes the computations among processes in a manner similar to the way that domain decomposition distributes the data. Often, the functional decomposition is viewed as a pipelining process since many different internal tasks are executed simultaneously. The master-worker method uses one process as a manager who is responsible for creating and managing the tasks for worker processes.

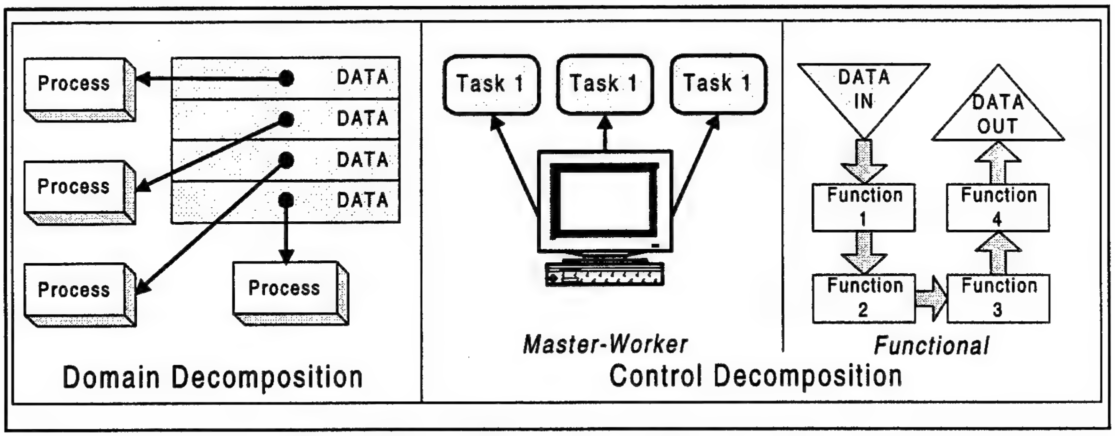


Figure 2.5.3 : Domain and Control Decomposition
This figure presents an alternative classification scheme for parallel decomposition from that of Carriero and Gelernter.

Domain and control decomposition are illustrated in Figure 2.5.3. It is important to compare this approach with the paradigms by Carriero and Gelernter. The functional approach in this section is the same as the specialist parallelism. However, the differences between the two sets of decomposition methods lie in the other categories. While the agenda parallel paradigm encompasses both the domain decomposition and master-worker (control) categories introduced in this section, the result parallel paradigm is not represented at all in the present approach. A third category that is often considered a special case of domain decomposition is the perfectly parallel technique. Applicable only in cases where very little communication is involved, the processes may truly run in parallel without having to wait for any other process to finish or for data to become available. This technique comes the closest to achieving the theoretical speedup.

Yet another alternative supported by authors who promote the domain, control, and perfectly parallel decomposition is the object-oriented approach. It is not meant as a fourth category, but simply as another way of looking at the algorithm. Considered as a formalization of functional and domain decomposition techniques, the object-oriented approach must consider both the data structures and computation distribution. Sequentially, the object-oriented programming model is based on passive objects. An object becomes active only when it is called, and only one object at a time is allowed to be active [BAL90]. The object-oriented approach can readily be ported to parallel processing by executing the objects that are unrestricted by data conflicts. This allows more than one object to be active at a time since they do not need to wait to receive a message before they start execution and do not have to leave their active state until completion.

Unfortunately, practical parallel applications usually do not run efficiently by employing only one of these techniques to the entire program. More often than not, a hybrid of these approaches is needed to make an efficient parallel algorithm, in which parts of the algorithm are broken down according to the decomposition method that best suits a particular section of the algorithm. A program decomposed in this way is referred to as hybrid or layered decomposition.

Units of Parallelism

Another set of decomposition methods is revealed by considering the units of parallelism of a program, or different ways that parallelism may exist in a program [BAL90]. This approach, offered by H. E. Bal, describes how to find parallelism in a sequential program for distributed execution. When a program is executed sequentially, the entire program is the unit of parallelism. When the program is to be executed in parallel, the program can be decomposed into various grain sizes that represent the units of parallelism. The five levels of parallelism in the Bal approach are processes, objects, parallel statements, expressions, and clauses. Each unit defines a unique way of discovering parallelism and decomposing algorithms for distributed processing.

The largest unit of parallelism is a process, which is defined by Bal as a "logical processor that executes code sequentially and has its own state and data" [BAL90]. Processes may be created implicitly by the compiler or explicitly by the programmer. Processes may communicate with each other, but care must be taken when implementing a process level job so that live processes do not attempt to communicate with terminated processes.

The object unit of parallelism is very similar to units in the domain decomposition. Unlike that defined by Ragsdale, the object-oriented approach by Bal is viewed as an alternative to the functional and procedural parallelism. The disadvantage of this view is that, in reality, the object-oriented technique is actually a hybrid of a variety of decomposition methods. Once the objects and the processes are defined, the algorithm fits naturally to parallel programming by finding parallelism in one of four ways [BAL90]:

1. Allow an object to be active without having received a message,
2. Allow the receiving object to continue execution after it returns its result,
3. Send messages to several objects at once, or
4. Allow the sender of a message to proceed in parallel with the receiver.

Methods 1 and 2 result in a program based on active objects, method 3 resembles the sequential execution model the most, and method 4 can exist by allowing multiple threads of execution.

When using parallel statements as the unit of parallelism, groups of statements are explicitly set off by identifiers that indicate whether the particular statements within the group are to be executed in parallel or executed sequentially. Generally, this produces a fine-grain program that is reasonably executed on shared-memory processors.

Functional parallelism, or parallelism at the expression level, consists of functions that act as processes with no side effects on other processes. Functional parallelism depends only on the input data. Simply stated, an expression forms a process whose input data are the operands and output is the result. Since the processes are distributed among the processors to exploit parallelism, the concept is similar to the functional decomposition encountered in previous sections. The difference is that the functional parallelism defined by Bal does not specify an execution order, and theoretically speaking, functions may be executed in any order or all in parallel. But it is well-known that this is a disastrous assumption! As with any parallel program, data dependencies must be examined and considered in the mapping of the program.

The final unit of parallelism, which is at the clause level, is used in logic programming, and is not addressed here. The reader is referred to [BAL90] and [TAKE92] for further information on parallel logic programming.

A major problem with using the units of parallelism to partition and decompose a program is that the each unit is largely language-dependent. In other words, often the language chosen must explicitly support the particular unit, and forming a hybrid approach can present a challenge likely to cancel out the benefits of using these guidelines. Figure 2.5.4 illustrates the units of parallelism.

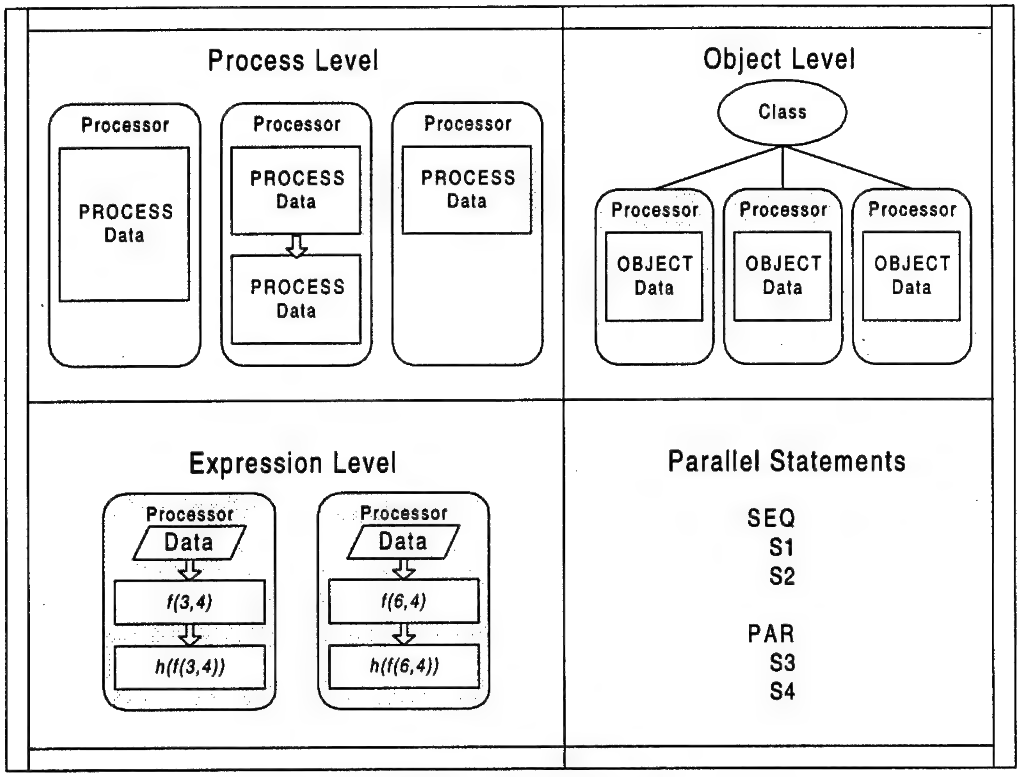


Figure 2.5.4 : Units of Parallelism

Four major units of parallelism introduced by Bal are processes, objects, expressions, and statements.

Divide-and-Conquer, Balanced Tree, Compression, and Doubling

The next two sets of decomposition methods concentrate more on the fact that most algorithms fall under a set of commonly-used topologies. These common sets have been extensively researched to find the best ways to parallelize the algorithms. The following two sections illustrate some of the resulting decomposition and partitioning strategies on representative sets of algorithms.

Divide-and-conquer (D&C), balanced binary tree, and compression comprise a set of design representative techniques [GIBB88, LEWI92, ZOMA95]. In addition, the concept of doubling is often included as a fourth technique. Divide-and-conquer, the most common of the three, divides the problem into a number of independent sub-problems that are handled recursively. This top-down approach is a good way to decompose a program because with grain packing, the steps can be combined to give varying degrees of granularity. Each leaf of the tree formed via D&C is a computation that is part of the overall problem. On the other hand, the balanced binary tree is a bottom-up approach that distributes the data and computations to different leaves. A computation is performed upon the data to obtain the next level of the tree. Figure 2.5.5 illustrates D&C and balanced tree. Two differences between them are: 1) the binary tree is balanced; and 2) binary tree is a bottom-up approach while D&C is a top-down approach. Some researchers consider the balance binary tree to be a special case of the D&C.

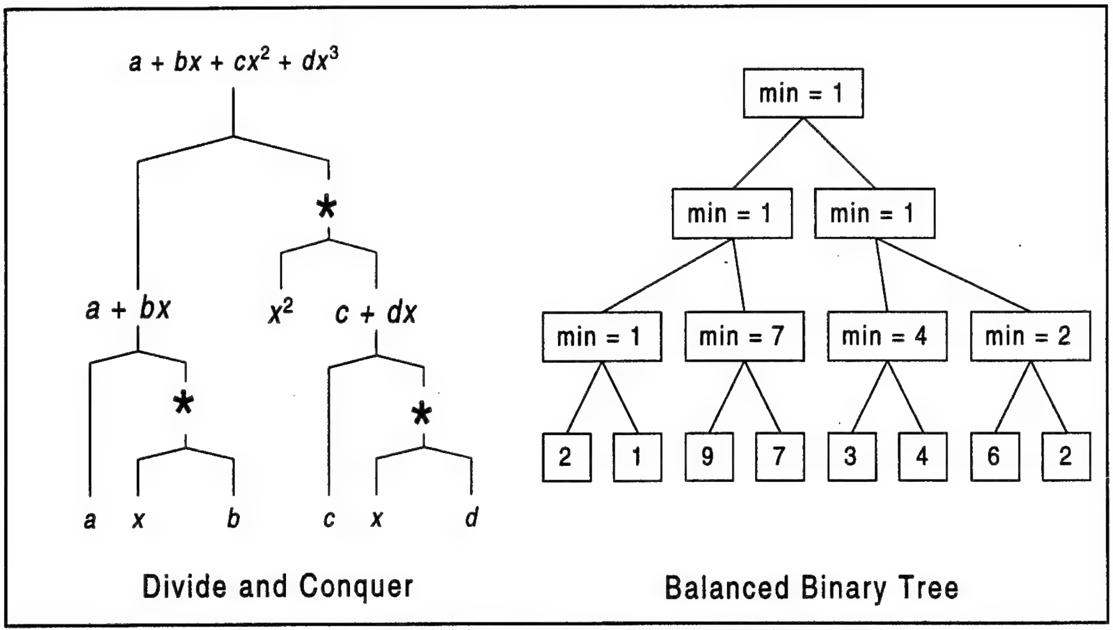


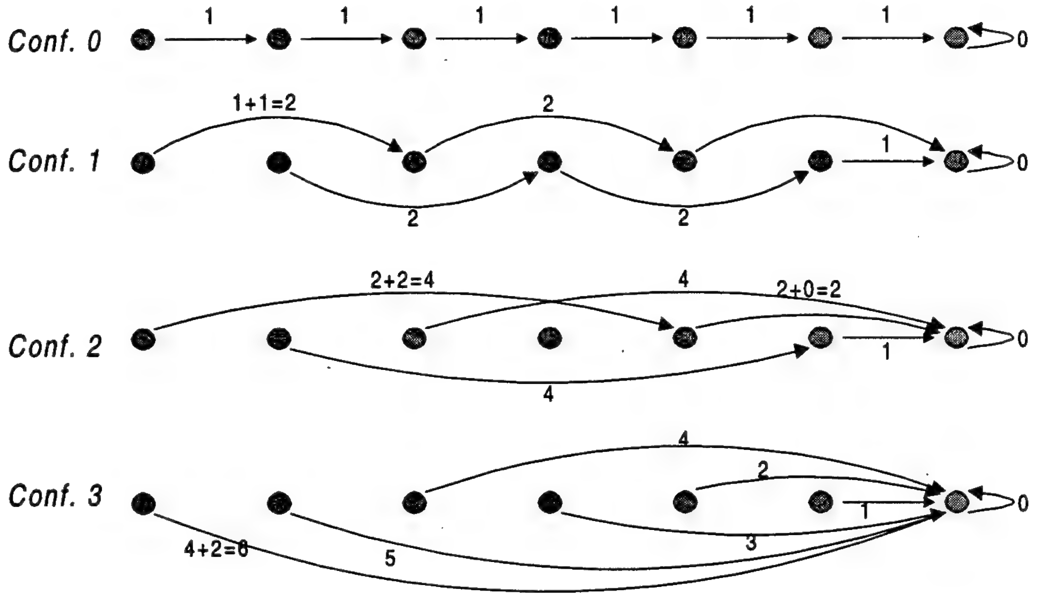
Figure 2.5.5 : Divide and Conquer and Balanced Binary Tree

The divide-and-conquer method is contrasted with the balanced binary tree in this figure. Notice how the right side of the D&C is not symmetrical with the left side.

Compression is similar to the balanced binary tree in that it aims to condense multiple data into a single or a few data items. For example, finding the maximum of a set of numbers may be accomplished by compression if for each pair of data at the bottom level, a single datum results. The comparison process continues until a single result is reached. This particular example is also a special case of the balanced binary tree in that data is distributed to the bottom of the leaves and the results are filtered up until the desired stopping point is reached. Compression is more flexible than balanced binary tree in that not only can it perform the same functions as those applied to a binary tree, but it also is very good at condensing graphing algorithms, such as creating one super-vertex instead of a number of smaller ones.

Doubling is the fourth technique considered. Normally applied to a list of elements, the idea behind doubling is to perform a specific function over all elements within a certain distance of each other. For example, a goal that can be achieved by doubling may be to rank elements, where the rank defines the position of the element in the list. The position of the element is found by counting the number of links that must be traversed to reach the first element of the list. If each element is assigned to a processor, then the number of elements equals the number of processors, and the rank of the element can be determined by the number of jumps required to reach the first processor. Doubling is best explained through the illustration and algorithm in Figure 2.5.6, in which the longest distance to reach the rightmost node (the first element and processor) is six. The figure is divided into four communication patterns that represent time progression. Configuration 0 is the initialization of the array given by the first group of algorithm statements, and the subsequent configurations illustrate the steps necessary to reach the last element and ensure that all elements are ranked. The algorithm given in the figure describes the steps necessary to achieve the list ranking. Because the longest distance is 6, the O(log *n*) algorithm requires 3 steps. The algorithm is described below the figure using the following variables:

k:	element and processor number
P(*k*):	the element pointed to by processor *k*
P(P(*k*)):	the element pointed to by the element pointed to by processor *k*
n:	the closest upward power of 2 from the number of elements
L:	number of elements
distance(*k*):	the distance of processor *k* from processor 0
rank(*k*):	rank of processor *k*



```
Algorithm {rank elements}
begin
# Conf 0:
# Initialize distance of elements based on whether it has a neighbor to the right or not
for all k∈L do in parallel:
        P(k)←next(k)
        if P(k)≠k then distance(k)←1
        else distance(k)←0

# Conf.1 - 3:
# If P(k) is not equal to what is pointed to by P(k), then increase distance of k by
# amount equal to the distance of P(k), and replaceP(k) by what is pointed to by P(k).
repeat log n times
        for all k∈L do in parallel:
                if P(k)≠P(P(k)) then
                        distance(k)←distance(k)+distance(P(k))
                        P(k)←P(P(k))

#assign rank based on final distance of k
for all k∈L do in parallel:
        rank(k)←distance(k)
end program
```

Figure 2.5.6 : Doubling [GIBB88]

The schematic and the following algorithm depict the doubling technique.

PRAM Models

The parallel random access machine (PRAM) model provides a way for the user to visualize parallelism in a parallel algorithm. The PRAM model, developed by Fortune and Wyllie in 1978, models an ideal parallel computer with shared memory and no overhead or communication costs [FORT78]. Figure 2.5.7 illustrates the PRAM model.

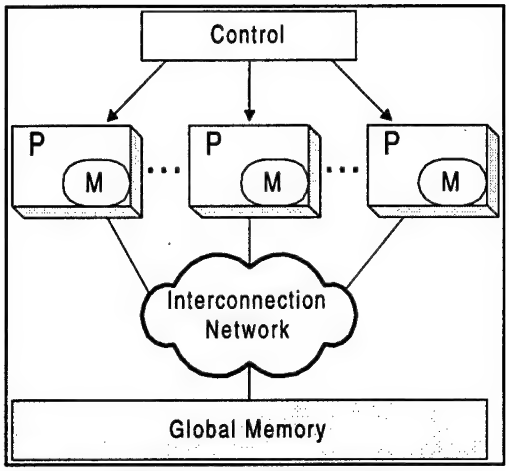


Figure 2.5.7 : PRAM Model

The PRAM model is depicted as a collection of processing elements which uses shared memory and has little communication overhead.

Although best suited to model tightly-coupled computers, the PRAM algorithm can give the designer insight on a variety of ways in which the program theoretically may be executed. The model is a valuable step in the development of a parallel program. There are four variations of the PRAM model, with each variation describing whether the processors in the model can read and/or write concurrently. They are:

1. EREW - exclusive read and exclusive write
2. CREW - concurrent read and exclusive write
3. ERCW - exclusive read and concurrent write
4. CRCW - concurrent read and concurrent write

Through extensive study of the PRAM model, new design techniques and many basic algorithms have been developed by which more elaborate algorithms may be based [GIBB88, HWAN93, QUIN94, ZOMA96]. These algorithms are designed to act as guides along the development of a parallel program. The most popular algorithms are briefly described in the next few sections.

Computing Prefix Sums. Also known as parallel prefixes or scans, the prefix sums problem is described best by the following [QUIN94]:

> *Given a set of* n *values* $a_1, a_2, ..., a_n$, *and an associative operation* $\oplus$, *the prefix sums problem is to compute the* n *quantities:*
>
> a_1
> $a_1 \oplus a_2$
> $a_1 \oplus a_2 \oplus a_3$
> ...
> $a_1 \oplus a_2 \oplus a_3 \oplus ... \oplus a_n$

The prefix sums has many applications. For example, it can be used to sum an array of numbers or to categorize a list of elements into subgroups without losing the original order of the elements within the categories.

List Ranking. Also known as the suffix sums problem, list ranking is the same as doubling introduced in the previous section. If there is a list of numbers each with location *i*, that location can be computed depending on the distance traveled to reach the last node.

Parallel Reduction. This is the same as the bottom-up balanced binary tree mentioned previously. One common example of parallel reduction is parallel summation, which is one of the functions required in the delay-and-sum beamforming algorithm that is simulated for this research. The elements of the array become the bottom leaves of the algorithm, and each level of the tree is the summation of the lower two leaves.

Pre-order Tree Traversal. This is a relatively complex algorithm presented for graph problems. It uses a structured tree whose edges are traversed only once in each direction. Though directed graphs are known to not be amenable to parallel computing, this offers an approach that comes close by matching the number of processes with the number of unidirectional branches on the tree. For example, Figure 2.5.8 illustrates the four phases of a tree traversal algorithm with eight nodes. First, a singly-linked list is constructed with each element being a path on the tree. Each element in the list is then given a weight depending on whether the travel direction was up (0) or down (1). The up-direction is assigned 0 because the node count does not increase as the tree is traveled back up the links. Third, the rank of the list element is computed by summing the number of 1's encountered in the linked list when traveling the list from the node in question to the last element of the list. Finally, for each downward traversal the rank is assigned.

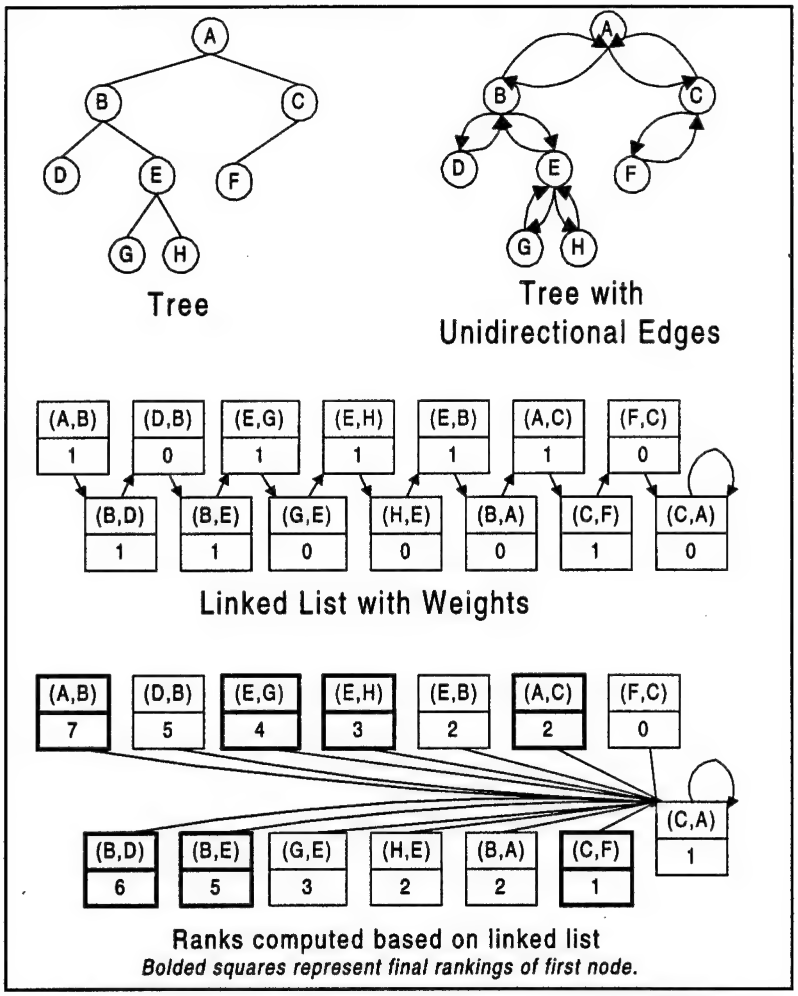


Figure 2.5.8 : Tree Traversal PRAM Algorithm [QUIN94]

The flow diagrams shown above illustrate the 4 phases of the tree traversal PRAM algorithm for 8 nodes.

Merging Lists. The final PRAM algorithm is the merging of two sorted lists. There are a number of ways this task can be implemented. One such method is Batcher's method, which uses a binary tree to merge the lists [GIBB88].

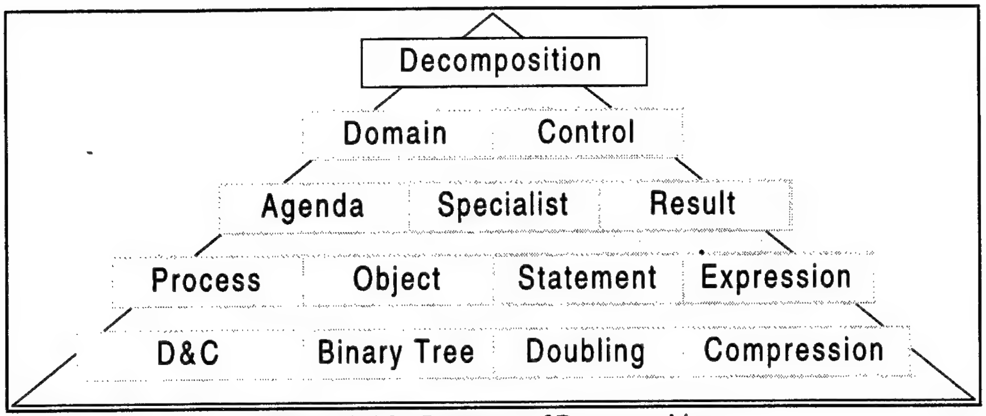


Figure 2.5.9 : Summary of Decomposition

This summary of decomposition methods and their classifications is based on all the methods discussed in the above section.

Figure 2.5.9 summarizes all the decomposition methods discussed in this section. There are a multitude of other methods that have not been cited, but most follow closely with one of these or with a mixture of these decomposition methods. The given sets provide a fundamental set of decomposition methods.

2.5.2 Total Program Design

Now that a foundation of decomposition techniques has been presented, it is important to investigate the role that algorithm decomposition provides in the overall process of parallel program design. Perhaps the most thorough method in the literature for parallel programming design is the PCAM (partitioning, communication, agglomeration, and mapping) model introduced by Ian Foster [FOST95]. The four stages of the PCAM model address all aspects of the parallel program, starting from machine-independent examination to actually mapping the program onto the actual processors to be used. The following is a brief description of the four stages of the PCAM model illustrated in Figure 2.5.10.

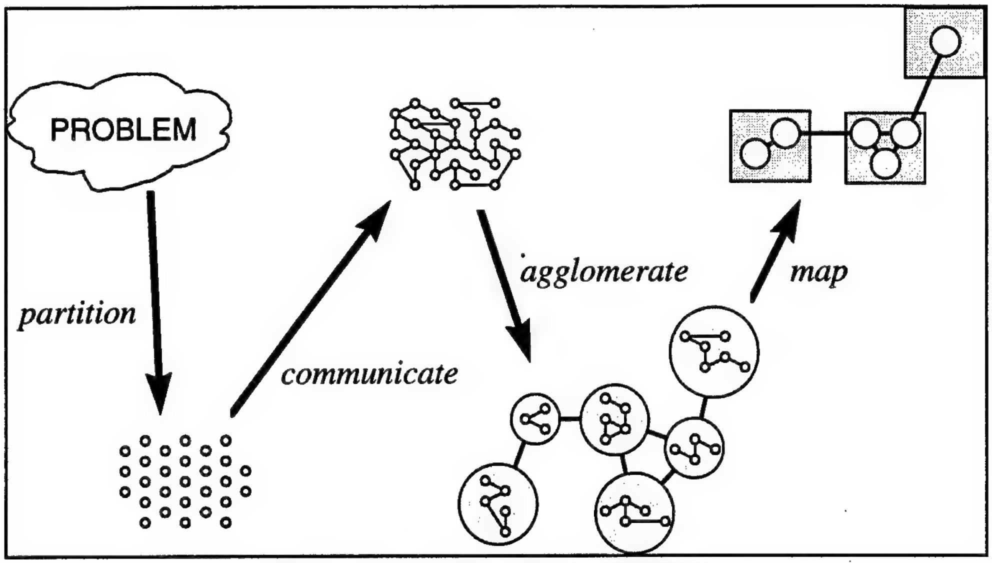


Figure 2.5.10 : PCAM Model [FOST95]

The 4 stages of the PCAM model include partitioning, communicating, agglomerating, and mapping.

Partitioning

The first phase of program development is the partitioning of the algorithm. The objective at this stage is to decompose the algorithms into fine-grain processes to exploit parallelism, regardless of communication costs and architecture. Most of the techniques mentioned earlier in this chapter may be employed in this phase of the program design.

Communication

This phase focuses on the traffic patterns and communication required for each of the processes that were identified during the partitioning phase. This is an important lead-in to the next two phases because it reveals the communication versus computation requirements for the finer-grained algorithms derived in phase one. Once completed, the next step becomes intuitive.

Agglomeration

Agglomeration involves grouping the processes together with granularity knobs to provide greater efficiency in the design of the algorithm. Hwang calls this technique grain-packing, where the processes are the grains [HWAN93]. A granularity knob will keep the agglomeration more flexible in that the grouping is not quite as strict. It is easy for the programmer to change granularity without making major changes to the program. The three goals of agglomeration include [FOST95]:

1. reduce communication by increasing granularity,
2. retain flexibility with respect to scalability, and
3. reduce software engineering costs.

Mapping

The final stage of the design process is the mapping of the processes onto the target architecture in a way that minimizes execution time. Often, this stage is fused with agglomeration by considering the processor configuration while performing the agglomeration. In this way, the agglomeration phase is more useful because all process groups that are created can be matched to a specific architecture. Many researchers have studied mapping techniques extensively and have written entire books on mapping alone [FOST95, PAIG93, SARK89]. However, there are some general representations used to govern mapping onto parallel processors. Two basic rules are: 1) place tasks that are able to execute in parallel on the different processors; and 2) place tasks with high communication among them on the same processor. There are two approaches to mapping algorithms: dynamic and static. In the dynamic approach, the mapping is determined at compile-time by complex algorithms that detect data conflicts and high-cost communication. In the static approach, the processes are explicitly stated by the programmer. Examples of some mapping algorithms include recursive bisection, probabilistic methods, cyclic mappings, and manager-worker [FOST95].

2.5.3 The Final Approach: Decomposition and Algorithm Development

The PCAM model introduced by Foster presents a thorough design method for creating modern parallel programs because of its flexibility. It used to be that most researchers only had access to particular architectures, and thus the program was designed around that architecture only. However, with the increasing popularity of parallel processing and discovery of more applications that need parallel processing, parallel computers are becoming more common, and it is likely that algorithms will be implemented on a number of different architectures. By using Foster's design method, only the mapping step and possibly the agglomeration step will need to be re-examined extensively when a change of architecture is required. All the work on the partitioning, communication, and some of the agglomeration with granularity knobs will be completed because each of those steps were done independent of the target architecture. For this project, this is an important point because parallel processing for beamforming applications is a fairly new field of study, and the architectures used today may very well change tomorrow. For these reasons, Foster's program design method is adapted with minor additions for designing the beamforming algorithms with phase one, the partitioning process, viewed from a PRAM perspective.

With the decomposition set used in this research defined, certain strategies can be identified for distributing the data in any one of the three techniques in the set and to act as sub-guidelines to accomplish the decomposition. These strategies are divided into two categories. The first category considers what happens to the data in each of the decomposition methods. Three methods as defined by Carriero and Gelernter are the live data structure, the distributed data structure, and message passing. The second category defines a general set of common techniques that may be applied to the division of the tasks instead of the data. Since many parallel algorithms are based on tree structures, the divide-and-conquer method and balanced binary tree are two recognized strategies of achieving task distribution. It is important to realize why these two categories are called strategies instead of actual decomposition techniques. They are meant to set examples for the decomposition, but by no means must all algorithms fall into one of the strategies. In this way, the flexibility of designing parallel programs is preserved, yet a systematic method by which parallel algorithms may be derived is employed. Table 2.5.1 gives a summary of the design process adapted.

Step	Name	Goals	Notes
1	**Partitioning**	• Expose parallelism in sequential algorithm. • Identify several alternative partitions. • Define comparable-sized tasks to assist in load balancing at later stages. • Avoid redundant computation and storage requirements	*Decomposition:* Domain Functional Master-worker *Data Strategies:* Live data structure Distributed data structure Message passing *Task Strategies:* Divide-and-conquer Balanced Binary Tree
2	**Communication**	• Optimize performance by tuning communication • Organize communication to allow parallel execution	*Communication Patterns:* Local/global Structured/unstructured Static/dynamic Synchronous/asynchronous
3	**Agglomeration**	• Reduce communication costs through grain-packing • Preserve flexibility with respect to mapping and scalability • Limit software engineering costs caused by intensive communication requirements	*Granularity* Avoid communication Increase locality Replicate computation Employ knobs for variance *Architecture* Determine optimum granularity
4	**Mapping**	Minimize execution time	• SPMD vs. dynamic task creation • Enable pointers for task location • Dynamic vs. static schemes

Table 2.5.1 : Parallel Program Design

Adaptation of PCAM model with modifications to the decomposition techniques.

2.6 UNIX Programming Techniques for Simulation

The Message-Passing Interface was the first of two major programming techniques that were used during the coding of the different parallel algorithms and their simulation, as discussed in Section 2.4. It allows different workstations to communicate via a common network. Both the SCI and TCP/IP versions of MPI were used with the testbed-based beamforming programs. Besides running parallel beamforming algorithms, MPI will also be used to connect multiple workstations during a distributed BONeS simulation. Because the programs contain simple SEND and RECV calls, an interface to any of the network architectures simulated in BONeS can be designed that mimics the low-level packet handling with MPI function calls. Plus, the MPI program will not need modification to be interfaced to the network simulator.

The second of these major techniques is the use of multithreading code, which is discussed in depth here. Multithreading allows a single program to do a number of tasks in parallel. In terms of simulation, these threads can be executing the code for different components in the scenario.

2.6.1 Multithreaded Programming

Threads can be simply defined as independent paths of execution which communicate through shared memory of the machine. To use threads, a programmer is forced to think about program execution in terms of multiple mostly-independent modules. Since these modules are independent, they can be run in parallel. In fact, multithreading is an easy method to exploit parallelism on a multiprocessor machine. Even when working on a single-processor system, which certainly cannot have more than one path of execution without time-sharing, a multithreaded program can still provide logical parallelism.

When a multithreaded program is run on a single-processor system, the kernel of the operating system is in charge of scheduling threads. Once a thread is scheduled, that thread executes until it encounters a condition causing a yield, which can be either an I/O request or a synchronization block. When a thread yields, the kernel chooses another thread to run out of a list of threads available in the scheduling loop. On a multiprocessor system, the operating system schedules as many threads to run at the same time as there are processors in the machine.

To write a multithreaded program, several functions must be used to manage the threads. The thread creation process consists of giving a function for the thread to execute, passing parameters to this function, and receiving a thread identification number. Once the child thread has been created, the parent thread may continue working, and both threads execute logically parallel. The thread identification number is used by the parent thread to keep track of the child and to join the child at a later time. Joining a thread means that the parent process will block at the join command until such time as the thread is done executing its function. At this time, the thread will be taken out of the scheduling process and the parent thread may continue its execution.

2.6.2 Synchronization Functions

Threads can use a number of mechanisms for synchronizing and communicating with each other, including mutual-exclusion locks (mutexes), semaphores, and conditionals. A mutex is an object in memory which may only be accessed by one thread at a time. When a thread "locks" a mutex, all other threads trying to lock the mutex will be blocked from further execution until such time as the thread which owns the mutex "unlocks" it [ROBB96].

This mechanism makes possible the mutual exclusion of resources; that is, it makes possible the assurance that only one thread has access to a particular resource. This is accomplished by making sure that any thread that wants access to the serial resource first locks a mutex created for that resource. Once the mutex is locked, that thread can be certain that it is the only thread using the resource, because any other thread will be waiting for the mutex to be released. This mechanism is depicted in Figure 2.6.1 by a set of machines that must be holding a key before they can access a disk. Once the disk access is done, the machine drops the key so that some other machines can contend for it.

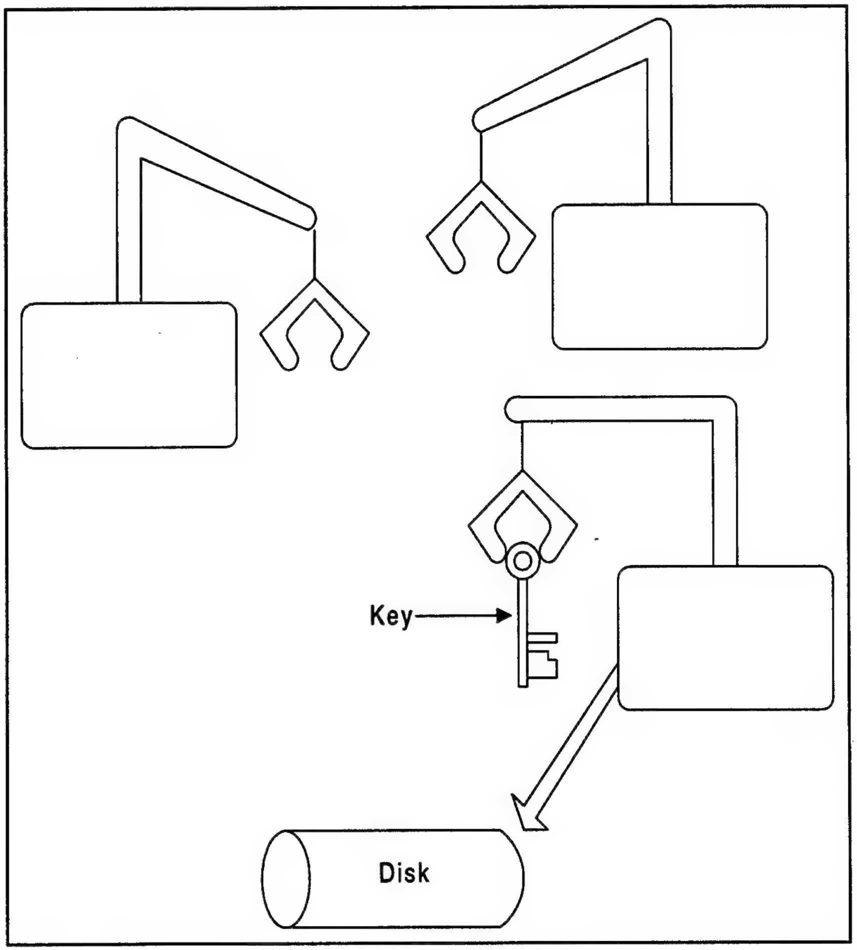


Figure 2.6.1 : Mutual Exclusion

This figure depicts a number of machines contending for a disk access which requires the mutex. They drop the key when done so that other machine may contend for it.

Another of the thread communication mechanisms is the semaphore, which is a memory location used for counting. Threads can increase the count of the semaphore by "posting" to the semaphore and can decrease the count by "waiting" on the semaphore. There are very important differences between a semaphore and a regular integer memory location used for counting. When multiple threads increment or decrement an integer in memory without any synchronization (such as a mutex), it is possible for more than one thread to access the memory at the same time and cause unexpected results. For example, if two threads both want to decrement the count by 1, both may read the memory at the same time and see it contains the value 10. Each thread decrements 10 by one to get 9 and places the value 9 in the memory location. Without mutual exclusion, the counter only ended up being decremented by one instead of by two. Semaphores fix this problem by making the change an atomic function. That is, between the time a thread reads the memory and when it writes the new value, nothing can interrupt the thread's process; no other thread can access the memory nor can the working thread be context-switched out [ROBB96].

Semaphores have an additional feature not available in a regular integer memory location. When a semaphore's count is zero, any thread trying to decrement the count blocks until some other thread in some other path of execution makes the count non-zero. This mechanism makes it possible for threads to signal each other. To do this, a semaphore is created and initialized to a count of zero. The thread which is expecting a signal waits for the semaphore, which causes a block since the semaphore is zero. The thread which wants to make the signal to the blocked thread then increments the semaphore to 1, which allows the waiting thread to successfully complete the decrement. To signal more than one thread, the semaphore simply needs to be posted more than once.

The semaphore mechanism also allows for a type of mutual exclusion in which a certain number under a limit are allowed to access a resource at the same time. To accomplish this, a semaphore is created and initialized to a number equal to the number of threads allowed to concurrently access a resource. When a thread wants access to the resource, it waits to decrement that semaphore, which causes a block if there are already the allowed number of threads accessing the resource. When a thread is done with the resource, it posts (or increments) the semaphore. This mechanism is illustrated in Figure 2.6.2 with a number of machines waiting on a limited number of keys to enter the room. When a machine exits the room, it drops the key on the floor so that all other machines know there is a new vacancy.

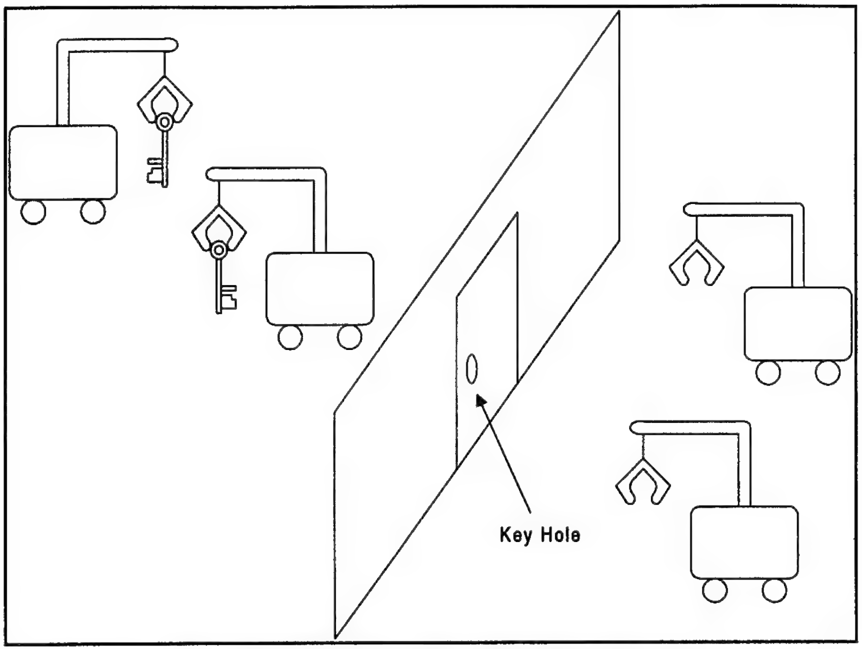


Figure 2.6.2 : Semaphores for Resource Exclusion

This figure depicts a number of machines trying to enter a room. There are multiple keys so that more than one machine can enter.

Another method of synchronizing threads is the condition mechanism. Rather than waiting for a mutex to become unlocked or a semaphore to become non-zero, a thread can wait for any variable to satisfy a standard logic condition. A thread may wait for some other thread to change a variable to become more than some value [ROBB96]. The conditional wait mechanism provides considerable flexibility in how threads synchronize with each other. For example, several threads may need to wait for a parent thread to increase an iteration counter before proceeding. To accomplish this, the threads can make a local copy of the current iteration counter. They then wait for a global counter from the parent to become greater than their local counter. When it does, the threads know they can continue with the next iteration. Without using the condition mechanism, the parent thread would have needed to signal the several threads to continue by posting several semaphores, on which the threads would have needed to wait.

2.6.3 Thread Implementations

One of the thread implementations used in the project is provided by the Solaris operating system. As is common with many commercial thread implementation packages, the operating system kernel is heavily involved in the scheduling and synchronization of threads. When a thread comes to a synchronization block, the kernel executes a context-switch in order to schedule another thread, which may then execute until it blocks or until the time quantum for the entire program expires. Solaris provides complex features to its mutexes and semaphores, including fairness and queuing order. An implementation that is unfair makes possible a situation in which a thread is waiting for a mutex to unlock (or a semaphore to become non-zero), but when it does become available, some other thread always gains access to it before the waiting thread has a chance. A fair implementation assures that a waiting thread will eventually get the resource no matter how quickly other threads can sneak in to grab it. The Solaris implementation also supports the ordering of threads waiting for mutexes and semaphores. For each mutex or semaphore, the implementation keeps a queue of those threads waiting for it. The kernel makes sure, without coding from the programmer, that the threads are scheduled to gain access to the resource in the order in which the waits were issued. The Solaris implementation also provides conditional waiting by creating special condition variables. Furthermore, a mutex must be initialized and associated with the condition variable. When a thread is waiting for a condition involving one of these variables to become true, and some other thread changes that variable, the kernel wakes up all the waiting threads with a signal. The waiting threads can then lock the associated mutex and check to see if the change has made their condition true. If the condition is still false, the thread releases the mutex and goes back to sleep. If the condition is true, then after releasing the mutex so that other threads can check the condition, the thread continues its execution.

Another thread implementation used in coding the parallel simulations of algorithms is the HCS thread library created by this research facility. The motivation for creating the thread library was that the thread functions in the Solaris implementation took more time than necessary, and in fact, using the threads added considerable overhead for fine-grained applications. For the purposes of this laboratory, many of the features in the Solaris implementation could be removed, such as queuing order, to lower the overhead of using threads. The HCS thread implementation is further sped up by moving the thread management and scheduling from the kernel into the user space.

The HCS thread implementation uses active waiting instead of signaling for synchronization blocks. That is, rather than having a thread go to sleep when it encounters a wait and needing a signal to revive it, the HCS implementation has the thread yield its time when it encounters a wait. After the thread scheduler has gone through all the other threads, it returns to the blocked thread, which then checks to see if conditions have changed and if it can continue, otherwise it again yields its time. The process continues in this manner until the thread finally gains access to its mutex or semaphore, at which time it continues execution, all without operating system intervention and the associated overhead. This type of synchronization has been coined "opportunistic polling" since the synchronization variables are checked whenever possible. The conditional variables with opportunistic polling now require no other signaling or special initialization. Also, a shared memory location can now be used for synchronization since operating system signaling is not necessary.

By using a multithreaded programming model with a robust inter-workstation communication technique, the organization and coordination of the simulated algorithms were taken care of, leaving development time to the optimization of the algorithms at hand. As the size and power of the target testbed increases, the programs, too, will scale since no notion of a fixed or maximum size has been introduced by these programming techniques. In other words, the completed code will continue to be operational as the workstations increase in performance and number.

2.7 Baseline Description

Specifying the baseline upon which this project intends to improve is an important part of the project. This section describes the sonar array that has been used as the baseline. Even though individual tasks may need to further define the baseline, such as a threaded-implementation baseline or a BONeS simulation baseline, the following description serves as the general foundation upon which these task-specific baselines are based.

The baseline is a large autonomous sonar array, such as that of the Rapidly Deployable Sonar Array (RDSA) program. It takes the form of a horizontal linear array of hundreds of data gathering nodes which do no computation. In this sense, the nodes of the baseline are often called "dumb" nodes to distinguish them from the nodes in this project's distributed multicomputer. Each dumb node is comprised of an audio transceiver to receive signals from the environment and just enough hardware to support buffering and sending the data. All nodes are connected via a unidirectional linear array fiber network. At one end of the array, a front-end processor is connected to the network. It is the job of the front-end processor to provide the computational engine for the entire array. Additionally, the node on the opposite end of the array from the front-end processor provides the master clock to the entire array. Figure 2.7.1 shows the representation of the baseline assumed for this project.

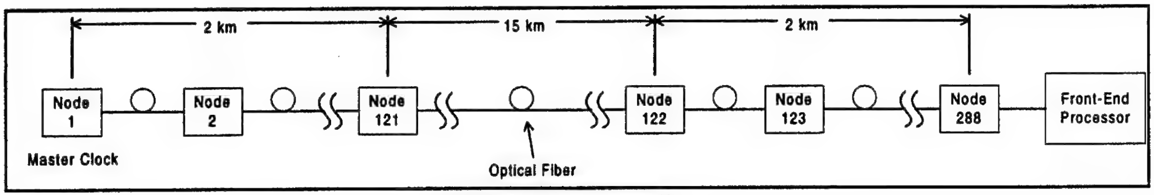


Figure 2.7.1 : Baseline

This figure depicts the general baseline used throughout this project.

The only job of the dumb nodes is to forward sampled data to the front-end processor by way of the unidirectional network. This is accomplished via a freight-train protocol, which can be pictured as a train with several empty cars. As the train moves from the most upstream dumb node to the front end, each node fills its specified car with data. In computer network terminology, the freight train is a constant-length packet. The packet flows through the network, initially with invalid data. As the packet flows through a node, that node's data is placed into the appropriate position in the packet, overwriting the invalid data. Each node is given a specific location in the packet so that no other node's data is overwritten by any node.

The data is analyzed by the end processor for desired information by some sonar array signal processing algorithm. It is this processor which is the cause of the concern over a single point of failure. If the end processor becomes unusable, none of the data produced by the array nodes can yield useful output, and the array stops working as a whole. No array node has the capability of taking over the computations for the front end.

With this baseline established, the researchers in the *HCS Research Laboratory* could begin the various tasks involved in finishing the project. The networks research proceeded by improving upon the unidirectional freight train of the baseline. The algorithms and software research proceeded by distributing the processing from the front-end processor. Comparisons to this baseline are made in each section to indicate performance and functional improvements.

3. Low-Power Hardware

Low-power hardware will be required in the design of an effective sonar array for an acceptable design. Through the reduction of power consumption in the array, many advantages may be achieved. Most notably, mission time will be increased. Other benefits of a reduced power system are a reduction in size and a reduction in weight. Today, devices can be found for almost any application which require very efficient operation. Improvements in technology brought about by the need for portable electronics have caused many semiconductor manufacturers to begin to produce devices which operate at extremely low power levels. Fortunately, technology continues to produce new methods of designing products which use even less power than the present standards. Of course, drawbacks in reducing the power consumption of a component may be found. In many cases, the performance of the product is reduced in order to allow it to operate efficiently. The task becomes finding the best fit for an application by reaching a compromise between the lowest possible power consumption and the highest possible performance.

3.1 Node Components

The critical requirement of the sonar array is the mission time of 30 days or 720 hours of continuous use. To attain this goal, each component in the array must be chosen carefully. The components in the array are microprocessors, analog-to-digital converters, random access memories (RAM), network interface (implemented by the processor or an ASIC), batteries, optical transceivers, and cabling. Furthermore, in the case of an optical link, an optical bypass switch capable of bypassing failed nodes will also be required. Figure 3.1.1 illustrates the major components that will be needed in the sonar array nodes.

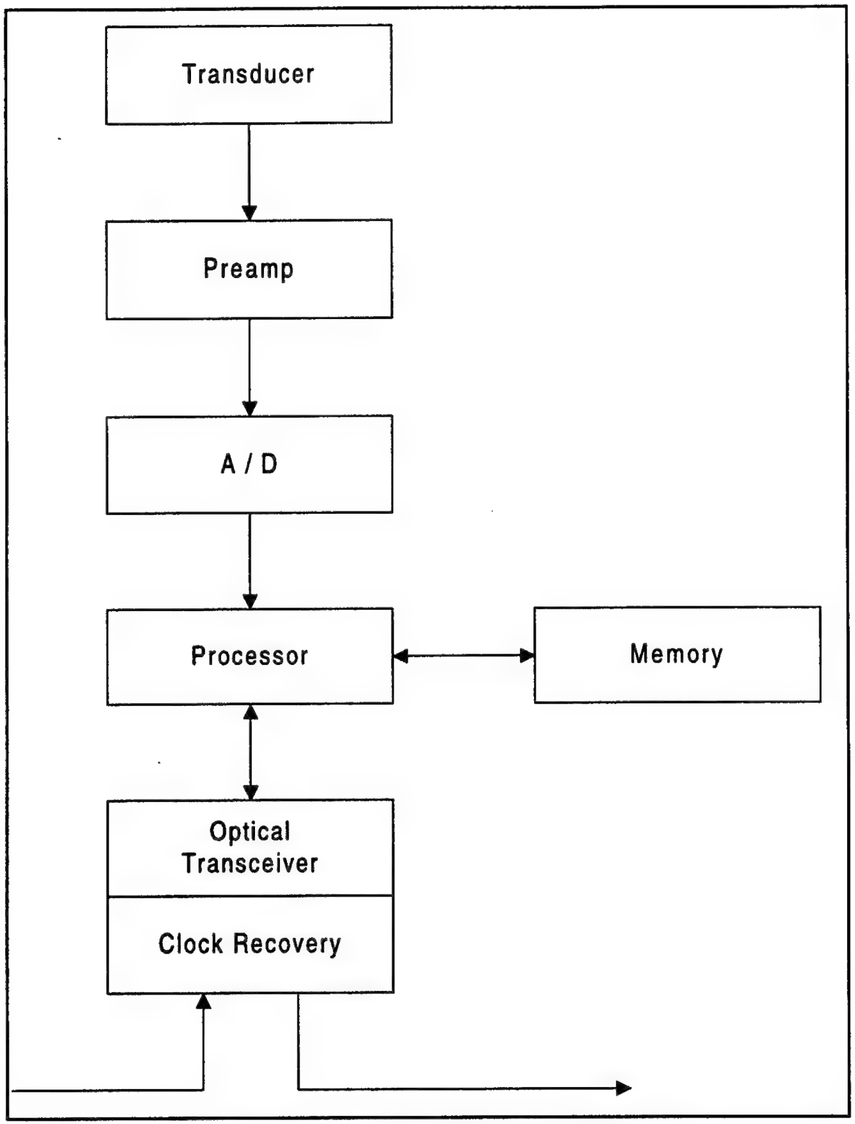


Figure 3.1.1 : Array Node Components
This diagram illustrates the different components needed in the sonar array nodes.

This chapter of the report will be an overview of some of the details involved in the selection of each device necessary for the sonar array as well as a trade-off analysis of the factors involved in making these selections.

3.1.1 Microprocessors

The microprocessor is the heart of the sonar array hardware design because it controls much of the logic of the network protocol as well as the execution of the beamforming algorithm and any fault-tolerant management protocol. Processors range greatly in their speeds, architecture, and capabilities, and it is hoped that for this project, a power efficient processor may be implemented which has enough speed to drive the software applications. In general, as the speed of a processor is increased, its power usage is also increased proportionally. Therefore, the optimal processor will have to be found which is able to maintain a good balance of power and speed. This also implies that the algorithms and implementations need to be whittled to the best efficiency to sustain livelihood of over 30 days of operation.

Microprocessor design has gone in a number of different directions, but one of the most important architectures for this particular project will be the Reduced Instruction Set Architecture (RISC). RISC processors employ pipelined and superscalar architectures to make chips more powerful. For example, the ARM7 processor core, from Advanced RISC Machines, is a device which employs a RISC main processor and state-of-the-art low-power techniques to achieve extremely low power while sustaining acceptable performance. Using a lower-than-standard voltage supply and a scaled-back clock speed allows power consumption to be drastically reduced. A typical processor based on the ARM7 core will run at 25MHz, complete 17 million instructions per second, and consume only 45mW of power. Typically ARM chips are used for low-power applications such as cellular communication or other portable technology, and are good candidates for a prototype COTS design for the sonar array.

One important question to be answered by this project will be whether the demands of the software will be too great for such a processor. Once rudimentary beamforming algorithms are designed which represent the order of complexity of the entire set of future algorithms, tests may be performed in order to determine the hardware requirements needed to perform the computations in real time. At this point, hardware selection may be made, and the requirements on the processor will determine what limits are imposed on the device and if a COTS microprocessor is a viable alternative to a custom ASIC design. Because there are many low-power microprocessors on the market, it is hoped that an efficient commercially available device may be found which will fulfill the specific needs of this project. The ARM7 is one such device that may satisfy the requirements once the power dissipation of the other devices is calculated.

Microcontrollers have also been studied for use as the main processing device in the sonar nodes. As opposed to microprocessors, microcontrollers are often aimed at embedded real-time applications such as digital signal processing. Due to the embedded nature of these applications, the devices often include several components on the chip that would normally need to be included and interfaced externally, such as memory and analog-to-digital converters. Furthermore, recent microcontrollers rival full-fledged microprocessors; many are based on RISC cores and support high-level languages [MAYE97]. Mitsubishi, NEC, and Toshiba make several chips aimed at the low-power market.

3.1.2 Analog-to-Digital Converters

Analog-to-digital converters (ADCs) receive the electrical output of the hydrophone transducer preamp and convert the analog voltage, related to the strength of the received sound wave, into a digital value which can be used and interpreted by the remainder of the hardware. The ADC selects an appropriate digital level from a finite number of selections in order to represent an analog signal which can assume a continuum of values. The ADC can perform this task, for example, by comparing the analog input to a set of reference levels which it has generated and which each correspond to a single digital output. In some architectures, a comparator exists for each possible digital output (used in very high speed conversion systems such as digital spectrum analyzers), while other architectures have allowed this number to be reduced at the expense of conversion speed. Delta-Sigma converters, which are one-bit serial converters, save power by reducing the number of comparators to one.

ADCs are implemented with a number of different architectures. These include pipelined, oversampled, charge distribution, and successive approximation. Of these types, the pipelined method seems to be the one which is most often implemented in applications requiring high speed and low power. The pipelined approach divides the problem, increasing speed through parallelism. This ADC uses two sets of comparators in order to determine the most significant bits (MSBs) separately from the least significant bits (LSBs) of the digital signal. The bits are then combined in a holding register and the entire digital signal is transmitted. By dividing the problem in two, speed and parallelism are increased, but the number of comparators in the device is also reduced, allowing for a more efficient design. The diagram below shows a simple model of a ten-bit pipelined ADC.

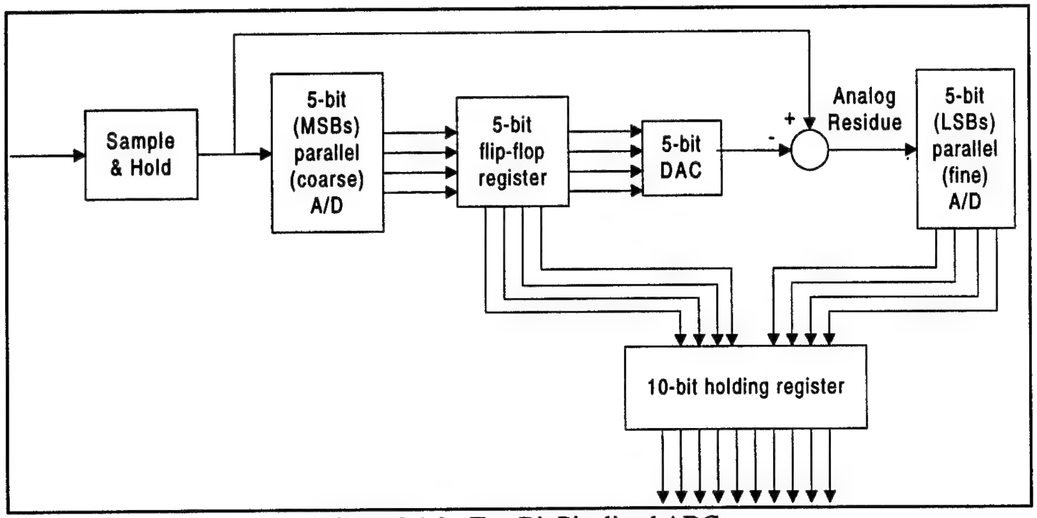


Figure 3.1.2 : Ten-Bit Pipelined ADC

This flowchart shows the data flow in a 10-bit pipelined analog-to-digital converter.

In the area of ADCs, many devices can be found which dissipate very little power. Some of these products may be found listed at the end of this section of the report. Due to the sampling theorem, signals of up to a kilohertz will require an ADC sampling rate of up to about 2.5 kHz. Even when sampling in the higher-frequency region, power consumption is small as compared to other devices such as RAM or the microprocessor. Power dissipation for a typical device is 1mW in sample mode with a power-down mode using only 25μW.

3.1.3 Random Access Memories

RAM is designed in two basic types, static RAM (SRAM) and dynamic RAM (DRAM), each having its advantages and disadvantages. Fortunately, present technology is allowing the differences between the two methodologies to be reduced. RAM will be selected based on the amount of code and data needed for this application and the power consumption of each available device. SRAMs use latches to store data without requiring refreshes. Each bit of storage requires four or five transistors. Although the design is very fast and efficient, it is also much less dense than DRAM. Therefore, SRAMs are the more efficient of the two architectures, but they typically are not produced with mass storage in mind because of limitations in size. Because of their speed, however, they are often used as cache RAM in computer systems. Evolving technology in SRAM development may allow for a stand-by mode in which the device's voltage source is reduced, allowing for a significant reduction in power when it is not in use.

DRAM design uses only one transistor as a switch, while storing bits of information on the capacitor gate of the transistor. Because of the discharging of the capacitor, the DRAM must be refreshed periodically (about every 15μs). This causes the design to be much less power efficient than SRAM, but also allows it a much greater density. Future technology in the development of DRAM should allow for a drastic reduction in the refresh rate of the capacitors used for data storage. With a considerable reduction in this rate, this device may become more practical for battery-powered applications such as the sonar array. Currently, however, power consumption may be too large for practical use. The plot below shows some of the trends in retention current, which relates to power consumption, of some typical SRAM and DRAM devices.

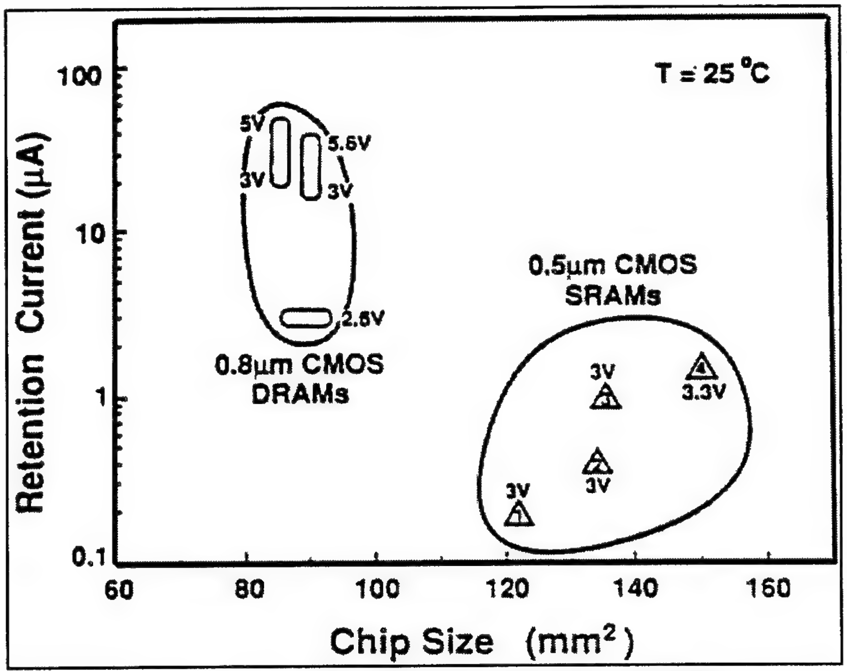
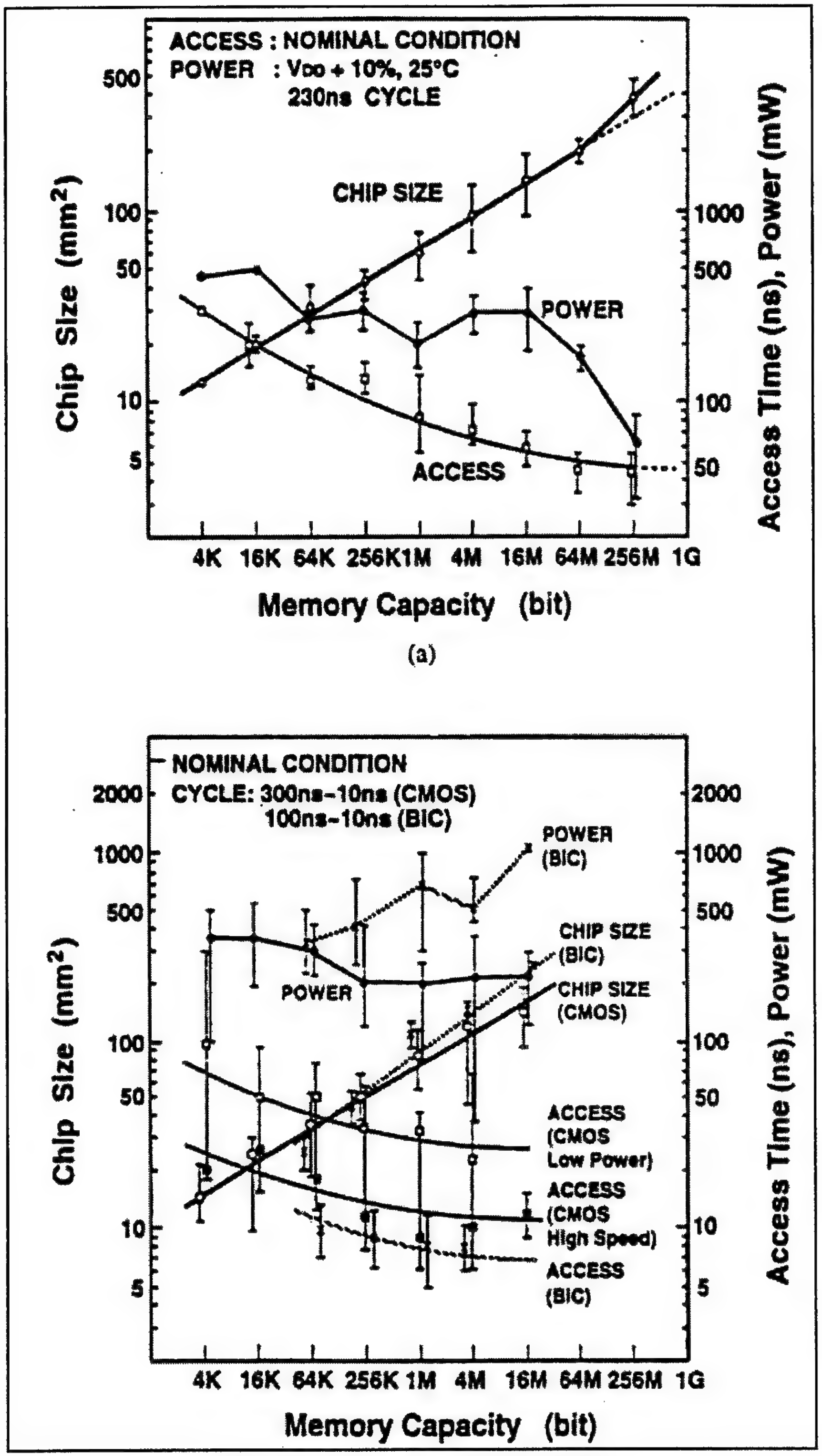


Figure 3.1.3 : Retention Current vs. Chip Size [ITOH95]
SRAMs have a larger chip size but a lower retention current, whereas DRAMs have a higher retention current and a smaller chip size.

Selection of the RAM to be used for the sonar array project will be based on a number of factors including the amount of memory needed, the power characteristics, and the speed needed by the device. The amount of memory needed will be determined through an analysis of the beamforming algorithm size and the quantity of data which will need to be stored by the array. Because it provides the best speed and power efficiency, static RAM is a good selection should the memory requirement not be too large. Dynamic RAM will have to be used if the density needs are great or if real-estate is a limiting factor. Of course, the power dissipation of each device must also be considered. Current trends in DRAM technology have allowed for a stand-by mode which operates on very little power (~1 mW). It is possible that with a given number of memory accesses in an application, use of DRAM will not become too much of a power burden because the device operates in the stand-by mode most of the time. RAM speed will be a determining factor only if the beamforming algorithm requires a great deal of speed from the device.

The plot below shows a relationship between power and speed and it should be noted that in most applications power actually is reduced with an increase in speed. This is because fast devices require more tightly-packed components, which also reduces power consumption. While SRAM is usually a good choice for speedy applications, there are also a number of DRAM devices on the market which offer faster access times than larger-capacity DRAM memories. All of these factors will be considered in the selection of RAM for this project. The chosen device will exhibit the proper amount of memory capacity while conserving power and demonstrating enough speed to run the chosen beamforming algorithms.



(a)

Figure 3.1.4 : Power vs. Speed [ITOH95]

The above graphs show how the power requirements increase as speed and memory density increase. The top chart shows DRAM performance and the bottom is SRAM.

3.1.4 Application-Specific Integrated Circuit

Another device which may be important to the design produced by this team is the Application-Specific Integrated Circuit (ASIC). The network protocol for this project may be implemented by either an ASIC created in-house or by a commercial-off-the-shelf (COTS) ASIC. One particular form of ASICs allows for user-programmable logic, letting a programmer design the operation of the device to particular needs rather than sending a complete design for fabrication. Traditional ASICs require a chip to be designed at the transistor level and then constructed by expensive photolithographic processes. Programmable logic allows for logic design with less complexity and a drastically reduced delivery time for a device. By using an ASIC for handling the network protocol, strain on the microprocessor can be reduced. ASICs would allow a good method of controlling the operation of the sonar array in an inexpensive manner if gate count and price limitations do not cause them to be impractical.

COTS ASICs are available on the market in a number of different forms which range in price as well as size and gate count. The gate count is an important metric because it limits the amount of logic which may be implemented by the ASIC. Certain applications with a great deal of complexity may require a large number of gates while others may need very few. Currently the largest ASICs contain around 600,000 gates, but emerging technology will allow for more gates and a lower price.

Programmable ASICs are currently designed in three basic categories. Programmable Logic Devices (PLDs) are very inexpensive ($2) but have a maximum gate count of 20-100 gates. FPGAs, Field Programmable Gate Arrays, show an increase in price ($20) but gate counts which range from 50-14,000. The final variety is the SPGA, System Programmable Gate Array, which incorporates memory with the FPGA design and raises the number of gates to around 600,000. Prices for SPGAs are in the range of $100. In order to determine the complexity of the required system, a protocol logic design will be constructed to determine which type of device will satisfy the complexity of the circuit yet remain feasible in terms of price and power consumption.

A major drawback to using COTS ASICs is that they are generally not low power. Without a major push in the commercial market for low-power ASIC devices, the available devices are very power-hungry. As such, COTS ASICs may prove to be poor candidates for use in this project; therefore, an in-house ASIC design is being pursued.

A preliminary protocol chip design will determine the usefulness of the ASIC in this application. It may be that the number of gates required for the network protocol is exceedingly large or that other problems will arise. By approaching the design from both the COTS and ASIC angles, the most appropriate implementation can be determined and will be used in the final design. Also, power requirements from both methods of the design will be determined. In ASICs, power requirements are generally increased in proportion to the number of gates implemented on the device, whereas in COTS devices, power consumption (though high) is usually fixed except by adjusting the clock speed or operating in special stand-by modes. A preliminary design will allow for a gate count of a protocol implementation and a test of whether power requirement's will make the ASIC a more practical alternative in this project.

3.1.5 Batteries

Powering each sonar array node will be a battery, powerful enough for an extended mission time, but not so bulky as to make it impractical. The selected battery should be able to fit within a size compatible with airborne deployment (e.g. C-cell size) and provide power to the node for at least 720 hours of operation (30 days).

One of the major factors in selecting a battery for use in a particular application is the type of electrochemical process used within the batteries. Some popular systems are listed below with their available energy capacities. It should be noted that the actual capacity shown on the chart is much lower than the theoretical capacity. This is because internal resistances and other non-ideal characteristics limit the amount of power which a battery can deliver. It should also be noted that both lithium compositions have a much greater capacity than do the other systems. Batteries using lithium chemical compositions have a number of advantages over other types including high energy density, high reliability, low weight, wide operating temperature ranges, long shelf life, and continuous power for extended periods of time [LIND95]. Lithium batteries have the added advantage of being in wide commercial use and therefore are cheap as compared to some exotic alternatives.

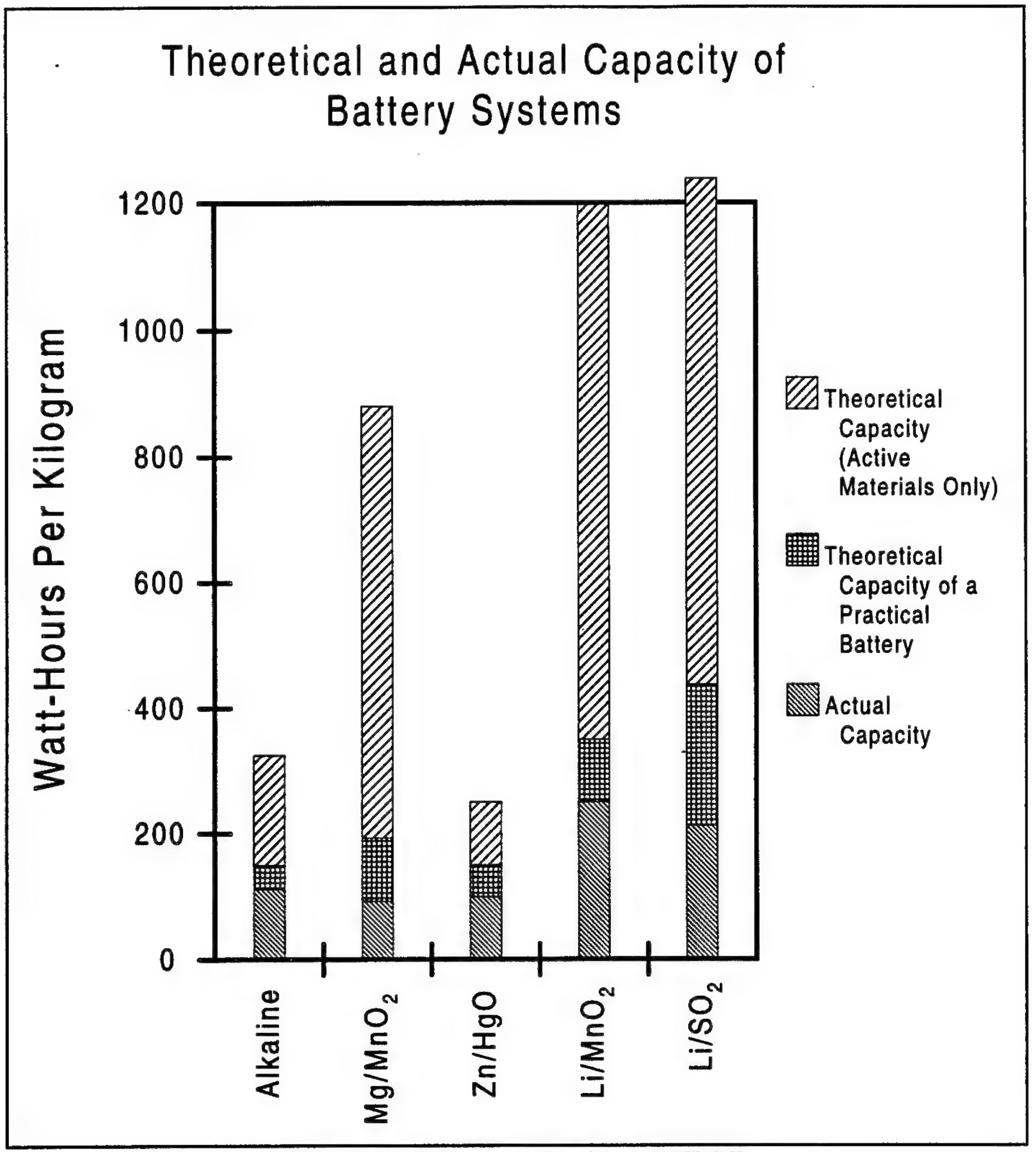


Figure 3.1.5 : Battery Capacities [LIND95]

This figure shows the theoretical and actual capacity of battery systems.

Batteries are typically designed as either a primary cell, a secondary cell, or a reserve. Primary cells are normally not rechargeable, have a high energy density, and are lightweight. Secondary cells can be recharged but normally have a lower energy density than a primary cell. A reserve battery may be a much smaller battery used for memory retention or device failure purposes [LIND95]. Recharging the battery will not be possible during the sonar array's mission, and although a secondary cell may be useful for memory retention or failure recovery, a primary battery better satisfies the requirements of the sonar array. Some specific examples of lithium battery specifications may be found at the end of this chapter. The batteries being considered can operate over a wide range of temperatures (-55C to +200C). They are perhaps the best-suited type of battery for long-term applications at low current.

3.1.6 Interconnect Medium

The selection of an interconnect medium for the sonar array is strongly dependent on the network topology and on the cost, size, and battery drain of the transceiver components used to implement the data link. Two types of media are available for interconnecting the telemetry nodes of the network: wire transmission line and fiber optic cable.

At the data rates required for the sonar array, both twisted pair and coaxial transmission line are suitable for distances up to several hundred feet. Over these distances an efficient differential current-sinking line driver will draw between 20 and 40 mA. This translates into a requirement for between 2 and 4 additional C-cell lithium batteries of the type described in Section 3.1.5. The additional cost of these batteries is estimated at between $20 and $40. In comparison, the fiber optic link described in the next section draws less than 5 mA. In order for fiber to be cost-competitive, the cost of the optical components used plus the additional cost, if any, of the fiber must be less than the cost of the additional batteries. If size and weight are the main concern then fiber will have a clear advantage. Furthermore, if the topology requires transmission over distances greater than a few hundred feet or projected data rates increase significantly, fiber will be the medium of choice.

Since the array is expected to extend over distances of about a kilometer, the cost of the medium is an important consideration. Over this distance multimode fiber is commercially available with a bandwidth of 400 MHz. Since multimode fiber costs less than single-mode and is also cheaper to align and connectorize, only multimode was considered for this analysis. A survey of wire and fiber suppliers revealed the interesting result that variations in cost of a particular cable type (e.g. shielded twisted pair) were greater than the average differences between types (i.e. twisted pair vs. fiber). In all cases the cost of the material was less than $1/ft. In practice, the cost of the transmission medium will be dominated by the cost of armoring the cable for submarine use. These results suggest that cost will not be a determining factor in the selection of transmission medium.

3.1.7 Optical Transceivers

The optical transceiver consists of the laser transmitter, laser driver, PIN photodiode, photodiode preamp, and bypass switch. In this section we describe experiments performed to evaluate the ultimate current drain, performance, and cost of these components.

The optical transmitter of choice for the sonar array is the Vertical Cavity Surface Emitting Laser (VCSEL) due to its extremely low drive current. VCSELs with threshold currents in the 1-2 mA range are readily available and devices with thresholds as low as 10 μA have been reported in the literature [KAJI97]. In contrast, edge-emitting lasers and light-emitting diodes typically require drive currents in the range of 20-100 mA. VCSELs have the added advantage of relatively simple packaging. Thus the system can be made at low cost and for long-time operation using batteries.

The sample VCSELs used in our experiments were prototypes supplied by Honeywell. These sample VCSELs are simply VCSEL "chips" mounted on standard TO-5 metal transistor can heads. A simple micropositioning stage was used to position a multimode fiber near the laser emitting surface. Since there is bonding wire attached to the laser surface, the fiber could not be put too close to the laser surface. However, because of the relatively low divergence of the laser beam from the VCSEL, the distance is not very critical. Also since the diameter of the core of the multimode fiber is 50 μm and that of the laser is about 10 μm (typical size 10 - 20 μm), the lateral alignment is not critical. Estimates for the coupling efficiency are around 50%. The coupling efficiency did not improve when imaging optics were tried.

The output of the laser at the other end of the fiber was measured with an Ando Type AQ-1135E optical power meter. The power output characteristic of one of the VCSELs as a function of the drive current is shown in Figure 3.1.6. The output characteristics of the second VCSEL were similar. The sample VCSELs were found to have a threshold current of about 3 mA and an output of 140 μW at 4 mA. The typical operating point is thus between 4 mA and 5 mA with an output of several hundred μW. The laser can be easily modulated without a bias current using a signal generator at speeds of 10 - 20 Mb/s. A typical pulsed output is shown in Figure 3.1.7. Note the optical output delay due to the rise time of the electrical pulses as well as the laser-stimulated emission delay.

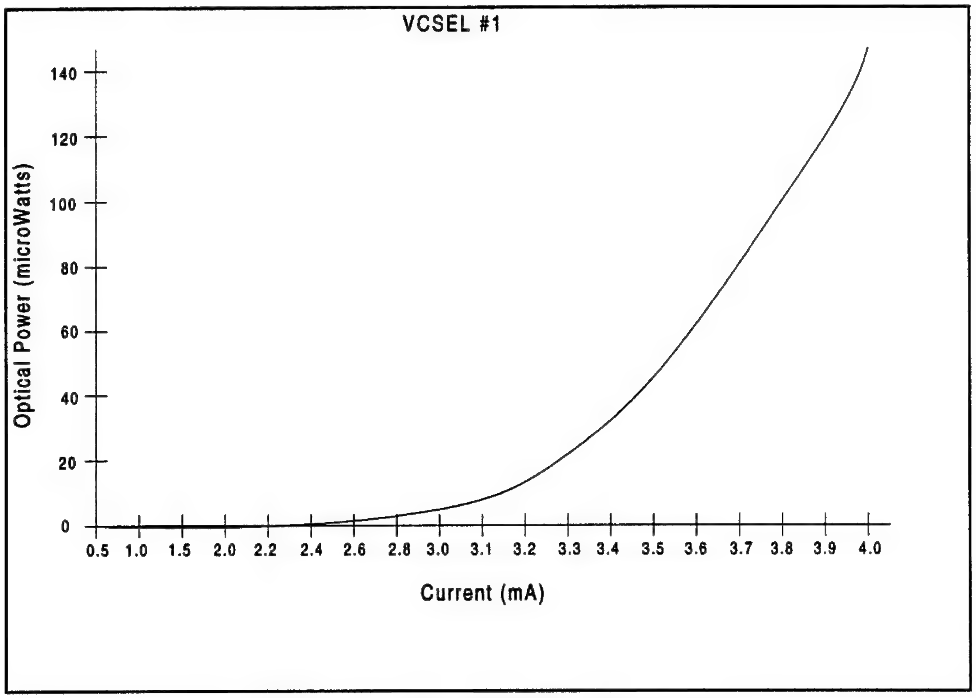


Figure 3.1.6 : Power Output vs. Drive Current

This figure shows the power output characteristic of a VCSEL as a function of the drive current.

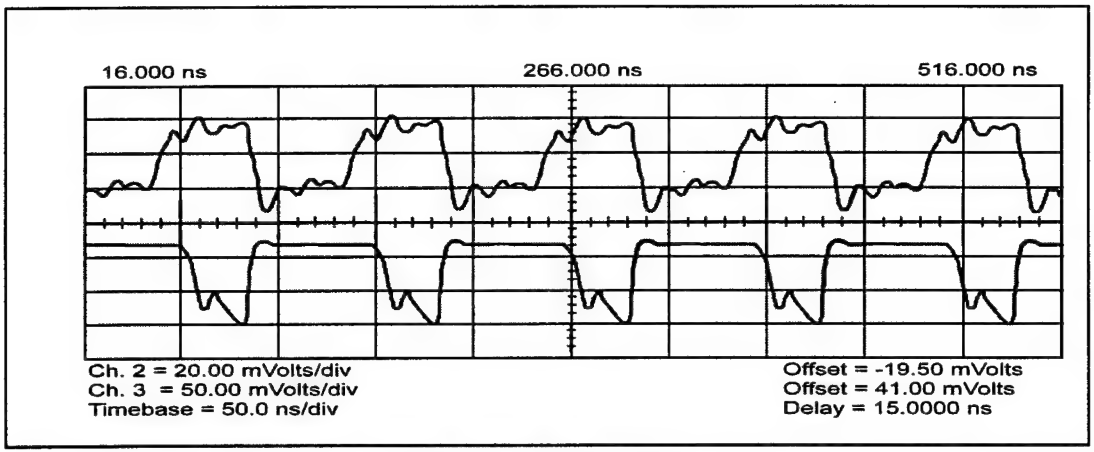


Figure 3.1.7 : Pulsed Output

This figure shows the pulsed output of a trial run using a signal generator.

Over the course of several weeks of testing, some aging of the VCSEL was noticed, i.e. a moderate increase of the threshold current. A systematic investigation of the aging of the VCSEL was not performed. It should be noted that the devices used in this study were early prototypes. Devices currently under development exhibit much lower thresholds (sub 100 μA), so increases in threshold even of several hundred percent will not be critical.

In order to achieve the lowest current drain possible, a custom drive circuit is required. The reason for this is that commercially available laser drivers are designed to achieve maximum optical power output at high frequencies; low-power operation is not a concern. In this study several custom driver circuits were evaluated. The important objectives are reliable operation and low cost. It was concluded that a bias current is not necessary at the data rates required for the sonar array. The driving current must be constant but can be easily changed to a different value for a different type of VCSEL. The current should also be independent of the supply voltage of the VCSEL. The exact circuit to be used in the system depends on several factors. An important consideration is that the driver can be integrated with the microprocessor. Another factor is whether or not the power supply (from a battery) is regulated. The terminal voltage of a battery can vary more than 30% over the defined battery capacity/lifespan.

A possible driver circuit is shown in Figure 3.1.8. If the voltage of the battery is always regulated (say at 3V), the transistor *Q4* is not needed. On the other hand, if we use a high efficiency (> 85%) voltage booster which does not regulate (boost) voltage until battery voltage is equal to or below the specified voltage (say 3V), then *Q4* is needed. One advantage of using a voltage booster is that it may actually increase the effective capacity of the battery. Only part of the circuit as shown in Figure 3.1.8 has been tested. The test circuit worked well to 10 Mb/s. Three types of microwave transistors have been tried in this circuit. They include Motorola MPS-911 (an old model that may no longer be available) and Bipolarics's BRF630-72, B12V114-18, B12V105-01, and BTA105-M2 (matched pair). The circuit worked well over a wide range of supply voltages. The unity-gain bandwidth for these transistors is between 5 and 10 GHz.

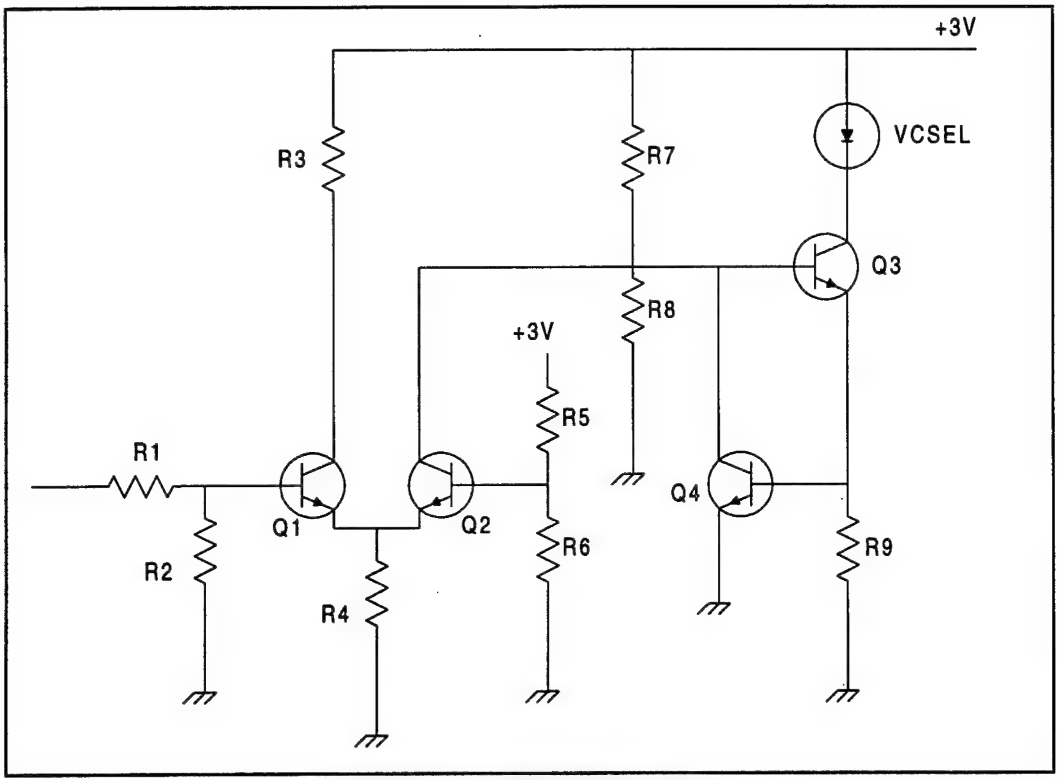


Figure 3.1.8 : VCSEL Driver Circuit
A possible driver circuit for the VCSEL device is shown here.

A PIN photodiode is the standard optical receiver for this type of application. The diode used is a Honeywell HFD3876-002. A low-power circuit is currently being developed for this device. The primary considerations in the design of the receiver/preamp are low power, low cost, and compatibility with the microprocessor. A commonly used transimpedance preamp is shown in Figure 3.1.9. The same transistors will most likely be used for the preamp so that it may be integrated with the laser driver in the future. The circuit is expected to draw slightly over 1 mA.

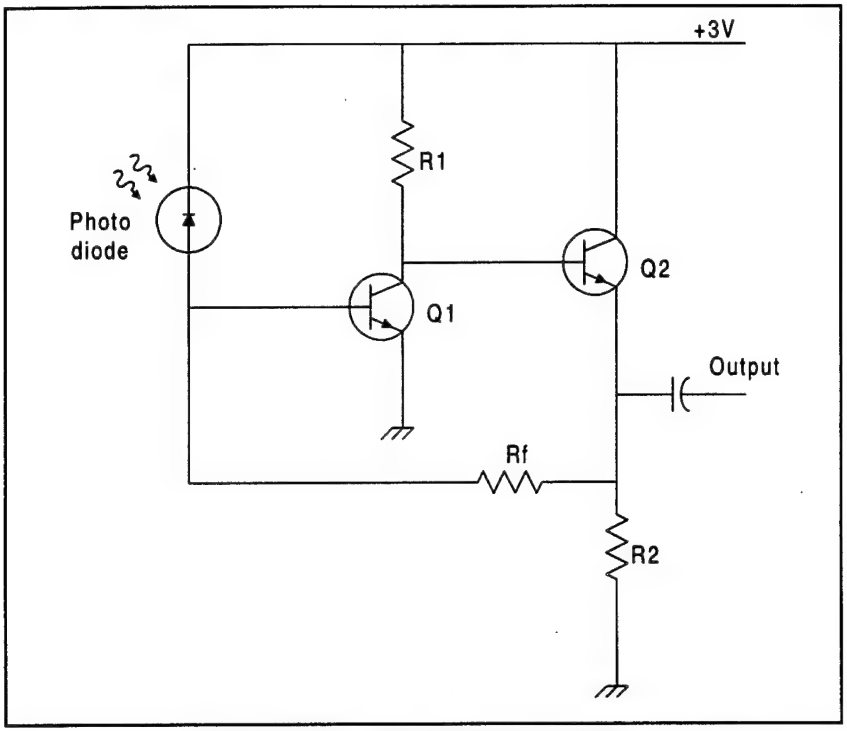


Figure 3.1.9 : A Transimpedance Preamp
This circuit diagram shows the layout for a transimpedance preamp.

In addition to the transmitter and receiver, a low-power optical bypass switch is needed to bypass failed nodes. Bypass switches may be either passive or active. A passive bypass is a simple fiber splitter and combiner joined back-to-back to send part of the optical input to the node and the rest around it, thereby bypassing the node if it fails. The bypassed signal is attenuated so as not to interfere with the signal from the node when it is operational. Typically the attenuation is about 10dB. For this reason only a small number of consecutive failed nodes may be bypassed before the accumulated attenuation decreases the signal beyond the detection range of the receiver.

An active bypass avoids this problem by using an active element such as a moving fiber or electro-optic switch. The tradeoff is that the active switch will be less reliable. Also, commercially available active switches are large, expensive, and quite power hungry. To investigate the possibility of developing a suitable active switch, an SBIR program was started in FY95 and a Phase I award was made to Fiber and Sensor Technologies, Inc. for the development of a reliable, low-cost, low-loss micropowered bypass switch. A prototype piezoelectric switch developed in Phase I demonstrated that a final Phase II version would be less than one cubic centimeter in size and dissipate less than 1 mW in the through state and no power in the bypass (fail-safe) state. The switch will be capable of bypassing 5 failed nodes. Estimated unit cost is about $25 in quantity. A Phase II award was made in late FY96.

As mentioned above, cost is a major consideration when selecting the interconnect medium. The cost of using fiber optics includes not only the added cost of the components but the cost of the packaging which can be quite high due to the need for careful alignment of the optical components. To a large extent the packaging problem can be minimized by using VCSELs coupled into multimode fiber and large area detectors. DARPA is currently funding a program in low-cost packaging of optical components and this should bring the cost down considerably. However, it is still likely that packaging costs will be higher for fiber optics in the foreseeable future.

The cost of an optical link can be further reduced by integrating the laser driver, photodiode preamp, and possibly the photodiode itself into a single Microwave Monolithic Integrated Circuit (MMIC). Since relatively few transistors are required for these functions the cost will be quite low. It should be noted that a wire link would also require a custom MMIC integrating the line driver and receiver to reduce cost and power. The cost of this MMIC would be comparable to that of the optical device.

The remaining component that must be considered in a cost estimate is the VCSEL source. VCSELs now available cost several hundred dollars. However, DARPA also has a program aimed at reducing the cost of these devices to the $5 range. If this program is successful the cost of a low-power optical interconnect for the sonar array may be comparable to that of a wire-based version, and the performance and size may become considerably better.

3.2 Design Techniques and Limitations

A number of factors typically play a role in the power efficiency of a device. Unfortunately, in most cases it seems that an increase in one desirable quality of a device will lead to a decrease in efficiency in some other desirable quality. In the design of the sonar array, it will be important to outline what factors are required, what the practical limits of device operation are, and which characteristics are needed and which are superfluous. For instance, with the ADCs discussed earlier, it was determined that sampling time was not required to be very fast. This allowed for finding a device which gives up sampling time in favor of an increased efficiency. On the other hand, with microprocessor selection, we must weigh the importance of speed against efficiency in order to determine the limits of an end product. Some major factors which will contribute to the selection of devices are performance, power, price, size, weight, and MTTF (mean time to failure).

Performance will be a key criteria in the selection of all devices. Even if very low-power devices can be found, will the devices operate properly? Will they degrade under strenuous operating conditions? With the microprocessor, this will be a key question because it is ultimately responsible for the operation of the array. As discussed earlier, each device has its own specific performance question. The goal of this project will be to reach the highest performance levels possible for each device while not causing the final design to suffer from any other problems such as price, weight, or unreliability. Through a determination of limits on the device qualities, compromises will be reached in the performance of each device which will deliver the highest level of operation while maintaining other necessary properties.

One of the most crucial design goals for the sonar array is that it use very little power, thus improving mission time and cost. There are many ideas to consider in searching for low-power devices. As technology continues to grow and computers continue to become smaller, many manufacturers have come to support low-power devices for use in hand-held and portable products. This has led to a number of advances in efficiency and made the search for products much simpler. One of the advances which has been made is a reduction of voltage supply levels required for digital devices. Because power is a quadratic function of the voltage supplied, this reduction has produced a tremendous increase in efficiency throughout the semiconductor market. Table 3.2.1 shows power reductions for a number of products based on a reduction in operating voltage. This advance has also pushed down the operating temperature and the die size required, allowing chips to be more tightly packed together on a board and to be much smaller. In general, advances such as this have allowed low-power devices to become available in almost every applicable area of the market.

Manufacturer	Part Type	Power Mode	5V Power (Max mW)	3.3V Power (Max mW)	Power Savings 3.3V vs. 5V
SRAMS					
Micron	256K x 8, 20ns	Operating	715	324	54%
		CMOS Stand-by	28	10	64%
IDT	256K x 8, 20ns	Operating	798	378	52%
		CMOS Stand-by	83	1.8	97%
DRAMs					
Micron	4M x 4, 80ns	Operating	495	180	44%
		BBU* (see below)	1.65	0.324	80%
IDT	256K x 8, 20ns	Operating	495	216	56%
		Self Refresh	0.72	0.36	50%
MICROPROCESSORS					
TI	DSP TMS320C5x	Typical Operating	13.8 mW per MHz	5.4 mW per MHz	61%
Motorola	DSP 56L002	Operating 40MHz	500	136	67%

Table 3.2.1 : A Comparison of 3.3V and 5.0V Memory Power Dissipation

The asterisk indicates Battery Backup current. This represents the DRAM operating at a CAS-BEFORE-RAS REFRESH at the slowest possible cycle time. In CAS-BEFORE-RAS, the address bus does not need to tell the device which cells to refresh. The device itself intelligently refreshes the capacitors.

Finding components with an appropriate price for this application will allow the sonar array to be produced efficiently and within budget. Again, an acceptable limit will have to be arrived upon in order for final selections to be made. If various devices are found from manufacturers, each will be analyzed based on price in order to determine the cost effectiveness of the product and if it is appropriate for this project. Hopefully, finding products within the appropriate price range will not create limitations in performance of the final design. Through current increases in technology, devices which have previously been overly expensive for this type of project have come into appropriate price ranges, but it is important that this project stay on the leading edge of technology so that the finished sonar array will be produced and manufactured for some time before it is outdated.

Device weight is another factor which must be considered in the sonar array design. While this will most likely not become a problem in any of the devices aside from the batteries, it should be considered in all cases. If the array is designed with a battery which is extremely heavy, it will be impractical to use. On the other hand, an increase in battery size and therefore weight will lead to an increase in mission time.

3.3 Device Summary

The ARM7 produced by Advanced RISC Machines is one of the processors under consideration for the node processors. It is rated at 18 mW at 10 MHz which is sufficiently low. However, it provides only the processor core thereby requiring additional hardware to be interfaced [ARM96]. An alternative is the μPD784214 produced by NEC. It is also rated at 18 mW at 10 MHz. In contrast to the ARM processor, the NEC chip is a microcontroller and it includes on-chip an 8-bit 8-channel ADC which would reduce the external hardware necessary for the node [HAYE97]. Other microcontrollers rated at 18mW at 10MHz include Mitsubishi's M16C and Mi6C. For lower power, another possibility is Toshiba, which makes the TMP93CW40F and TMP93CW46F microcontrollers. They are rated at 8 mW at 12.5 MHz. An on-chip 10-bit 8-channel ADC is included [HAYE97].

If external ADCs are determined to be necessary to the node design, a possibility is the AD7854 by Analog Devices. During the conversion process, only 1.3 mW of power is consumed, and this is reduced in stand-by mode to 25 μW. It performs at a rate of 10 kSps (kilo-samples per second).

Possibilities for memory include the MB814405D DRAM or the TC55V328J SRAM. The DRAM produced by Fujitsu (MB814405D) operates at a maximum power consumption of 385 mW and reduces to a maximum of 1.1 mW during stand-by mode. It has a 1M x 4-bit organization and a maximum access time of 30 ns. On the other hand, the SRAM (TC55V328J) produced by Toshiba consumes 231 mW during operation and 1 mW during stand-by. It has a 1M x 8-bit organization. With the use of low-power microcontrollers, their on-chip memory for program and data storage may make it possible to operate the main memory devices elsewhere on the node board in low-power, stand-by mode for extended intervals of time by taking advantage of spatial and temporal locality in the software.

Possibilities for batteries include the SL2770 by Newark Electronics, Distr. It is rated at 8.0 Amp-hours and has a mass of 52 grams. Another possibility is a Battery Engineering product, one of which is rated at 6.5 Amp-hours and has a mass of 52 grams.

4. Topology, Architecture, and Protocol Development

4.1 Network Topologies

4.1.1 Unidirectional Array

A unidirectional point-to-point network is an array in which each node, excluding the first and last, has one incoming connection and one outgoing connection. The stream is unidirectional and thus limits the protocol in many ways. The first limitation is that for a network of this structure to work, the traffic must be absolutely deterministic, that is non-stochastic. If random traffic is allowed any upstream node is likely to utilize the full bandwidth of the network, blocking nodes further downstream. The topology also limits the communication ability, making it impossible to construct an acknowledged-based protocol.

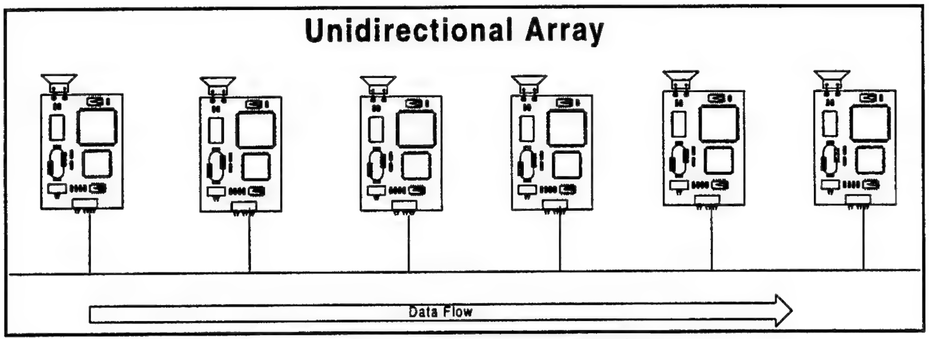


Figure 4.1.1 : Unidirectional Topology

In the unidirectional network, nodes are connected in series and data flows one direction.

The Medium Access Control (MAC) for a network of this sort could employ a few different techniques. The furthest node upstream may become the master and create a large packet, one so large each node has room to squeeze its particular data into the packet, or "freight train," as it passes. This is, in fact, the basic protocol structure used in the RDSA Baseline architecture. Another technique could employ an insertion buffer. The buffer allows random contention of the network without fear of collisions. If the insertion buffer is empty, a node may start to transmit a packet from its output buffer. If a packet were to arrive while the node is still transmitting its own data, the incoming data would simply begin to fill the insertion buffer.

The unidirectional array using either MAC protocol above is a very simple algorithm. So simple, in fact, that no management protocol is required to ensure fairness. It is inexpensive to implement because it requires only two network connections, one in and one out. It can become more fault-tolerant if the protocol includes a mechanism to bypass a failed node and re-calibrate the network to create a new network master. Its biggest weakness is probably its oversimplified topology. With such limited access to other nodes, the communication for many algorithms would be impossible.

4.1.2 Token Ring

The classic ring is constructed by single ported nodes communicating in a unidirectional manner over a closed loop. Unlike the unidirectional array, which is open at the ends, the ring is a fully-functional network in which each node may communicate with every other node by passing packets through intermediate nodes.

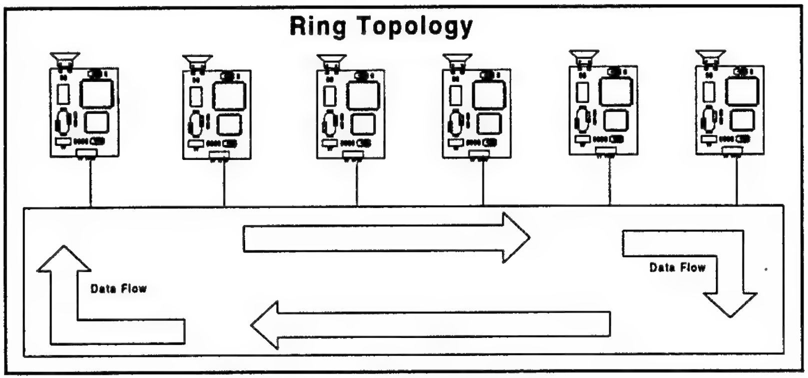


Figure 4.1.2 : Ring Topology

In the ring network, nodes are connected in series just as in the unidirectional except the ends are connected. The traffic is still in one direction.

Contention for the network of a Token Ring is based on movement of a "token." This packet circles the ring and each node may hold the token if it is passed to it. Thus, since there is always only one master node there is no contention for the network. The token is based on the IEEE 802 standard which includes fields for priority mode and priority reservation. Other fields were unneeded in the simple protocol proposed for this network. One way of interpreting the protocol for a token ring is to imagine that the node which is holding the token opens the connection from input to output. All other nodes close the connection and "snoop" the incoming packets to see whether the packet is addressed to them. The receiving node does not pull the packet from the ring, however. The node which sends the information will instead receive an echo of the packet which ensures that the data traversed the ring. If this echo is not received the node retries. This paradigm is very different from other ring protocols such as insertion ring in which the receiving node is in charge of stripping the packet off the network. Usually, bit stuffing is a requirement to delineate the token from data packets; however, since the communication will be encoded with a block or serial encoding scheme, special illegal characters may be reserved for control purposes such as a token.

The advantage of a ring over the unidirectional array is the full connectivity of the network. Theoretically, any algorithm may be mapped across a network of this sort. The question, however, may be one of efficiency. Its greatest weakness of the ring is its poor fault-tolerant nature. Any break in the ring or a bad node destroys the state of the system. However, it may be argued that it is no less fault-tolerant than a unidirectional array since a broken ring could theoretically be reinitialized as a unidirectional array. But, the overhead to ensure this degree of fault-tolerance would seem overbearing. Not only would the network protocol need to be adapted in this scenario but also the application would have to be able to support low connectivity or limited communication.

4.1.3 Insertion Ring

The topology of a register insertion ring is no different than the token ring. The only real difference is in the contention of the network. Register insertion is named after its most distinctive trait: the insertion buffer. Three buffers typify the MAC protocol: the insertion, input, and output buffers. The basic theory of an insertion ring is that every node has equal probability of writing to the network. In fact, an insertion ring could be compared to Carrier Sense Multiple Access with Collision Detection (CSMA/CD), more popularly known as Ethernet. In CSMA/CD, contention for the network is random. Each node has an equal opportunity to write to a common medium or bus. The problem with CSMA/CD is the topological nature in which each node competes. This results in collisions that require collision detection to manage the problem. The benefit of register insertion is its point-to-point topology. This eliminates collision, and allows for greater utilization of the network, perhaps 90% instead of 35%. One inherent drawback of an insertion ring is its ability to block nodes from sending. This is especially apparent in large networks of nodes. Management protocols must be integral to the network to ensure fairness for each node. Despite these challenges and limitations, the DPSA (Distributed Parallel Sonar Array) ring may exclude the overhead of these management protocols since the traffic imposed on the network will be deterministic, eliminating certain nodes from holding the network.

The strengths of the insertion ring are the same as for the aforementioned token rings. However, register insertion has the added advantage of better utilization with the traffic patterns intended for the network. This will allow for a slower network frequency which undoubtedly will show power advantages. Register insertion typically also has lower latencies than the Token Ring protocol. A testament to its ability is its use in the Scalable Coherent Interface (SCI) used in supercomputer technologies and cluster-based parallel machines. Its primary weakness is also shared with token ring: limited fault-tolerance.

4.1.4 Bidirectional Array

The bidirectional array shares features of both the unidirectional array and the insertion ring. The array could use any number of different protocols but the insertion buffer scheme seemed to be the most natural and easiest to implement. The bidirectional array could be connected by the use of two discrete paths, or to save cabling costs, the signals could be multiplexed and share the same link. Either way, a node cannot avoid the use of two I/O ports, one for traffic in each direction. Each layer, as will be shown below, is created by a symmetric system for each direction up until the network layer.

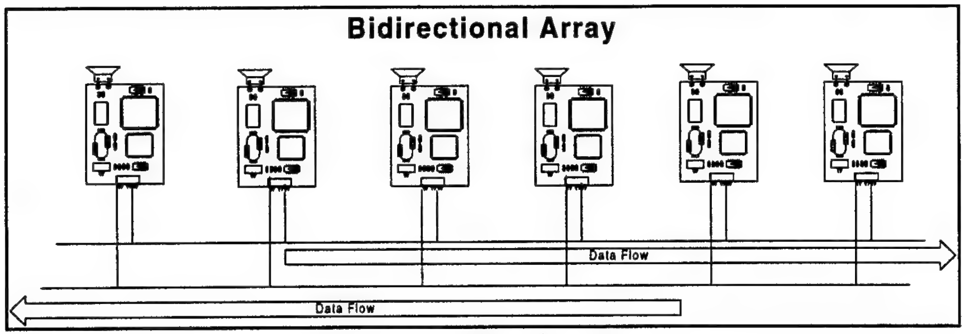


Figure 4.1.3 : Bidirectional Array

In the bidirectional array, nodes can communicate either up or down the array, but there is no end-to-end connection as in the ring topologies.

The MAC was constructed of two insertion buffers and two output buffers. The protocol could be used as a dual concentric insertion ring or a bidirectional array. The MAC was built by mirroring all of the buffers in the register insertion MAC except the input buffer. Only one input buffer is required and may be doubled in size if necessary to compensate for the doubling of the other buffers. As mentioned above when discussing the insertion buffer technique, each node has random contention in placing traffic on the network. This is not very successful when operating the networks with random traffic but again the traffic intended for this network is deterministic so no management protocol is needed for fairness in the MAC layer.

The bidirectional array has full connectivity and has the distinction of being the most fault tolerant. If a part of the network goes down, the rest of the network can reinitialize as a fully functional bidirectional array. Therefore, it is a much better alternative to the ring topology because of fault-tolerance and much better than the unidirectional array because of algorithmic complexity and flexibility.

The weaknesses of the bidirectional array are a matter of cost. Although, the topology can use a single cable via multiplexing, it still requires more hardware per node for dual ports and dual bypasses to eliminate faulty nodes. If multiplexed on a single link, the network will also have to run at twice the speed which will ensue some power costs as well. On the other hand, the bidirectional array is much more efficient than the ring due to the ability to traverse less hops when communicating (on average, half as many hops), which may increase efficiency enough to overtake the power disadvantages.

Unidirectional Bus Linear Array	
Advantages	**Disadvantages**
Lowest cost	Moderate fault tolerance
	Limited algorithm flexibility
Unidirectional Ring	
Advantages	**Disadvantages**
Full connectivity	Poor fault tolerance
Greater algorithm flexibility	Longest cabling runs
Bidirectional Linear Array	
Advantages	**Disadvantages**
Full connectivity	Twice the # of Tx/Rx, high cost
No additional fiber cost	Complex protocol
Greatest fault tolerance	
Greatest algorithm flexibility	

Table 4.1.1 : Topology Comparison

The advantages and disadvantages for all three topologies is given in the above table.

4.2 Network Modeling using BONeS

The Block-Oriented Network Simulator (BONeS), a commercial program created by the Alta Group, is the discrete-event driven simulation package used for all networking modeling simulation and analysis in this project. In this section we present an overview of this CAD tool. Its main features allow one to build a data structure (Data Structure Editor), operate on the data structures with blocks (Block Diagram Editor), execute hierarchical block models (Simulation Manager), and build output graphs (Post Processor). An event may be defined as an outcome such as the movement of a data structure into a node. Events are propagated through the network of structures in time slices. Events do not actually take up time until they hit a delay. The execution of the simulation stochastically traces events which occur in zero time until all events of the slice of time are consumed. The clock is then incremented and the next set of events are executed based on a set of scheduling rules. This is continued until the entire simulation time is completed. Events are not completely operated in stochastic sequences. Some primitives in BONeS give the user execution controls to ensure that certain events occur in a determined sequence although within the same time-slice. BONeS simulations are completely fabricated in C/C++ code from blocks of code called primitives. BONeS designers can use this to their advantage in creating new primitives, thus, are limited only by C/C++ and the operating system.

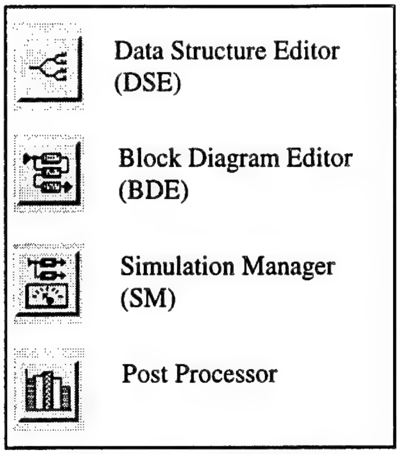


Figure 4.2.1 : BONeS Tools

The above icons represent the different tools in the BONeS designer program.

4.2.1 Data Structure Editor

The data structure is particularly interesting. Inheritance is a quality of each data structure shown in Figure 4.2.2. "Composite" may be called the "parent" of other structures below itself such as *Boxcar2* which may be called a "child." Children inherit characteristics from their supers (or parents); fields present in the parent will also be present in the children. This implies that children are aggregates of their parent, and that supers or parents are generalizations of the children. Aggregation describes when something is composed of subparts such as a house which is made up of a kitchen, living room, bedrooms, and bath. Generalization describes when items are grouped together such as a hammer, screwdriver, etc. being generalized as tools. To illustrate this, observe the data chart below taken right from the BONeS Data Structure Editor. Notice that the root of the tree is "Trigger" followed by "Root-Object." Every data structure created in BONeS is a generalization of these types. In other words, any data structure may act as a trigger or a root object. The question of aggregation is a more subtle one and may be described in more than one way. On one hand, supers are built up from aggregates, such as Boxcar2 which is built of *Token/Datas*, *Idles*, and *Fault_Tokens.*

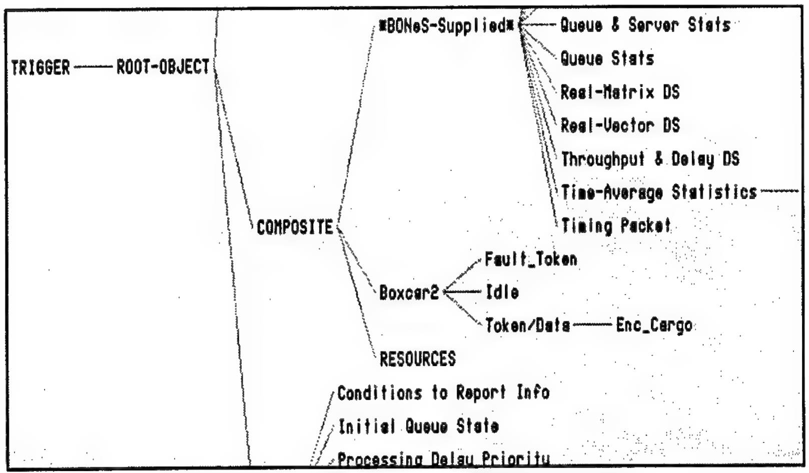
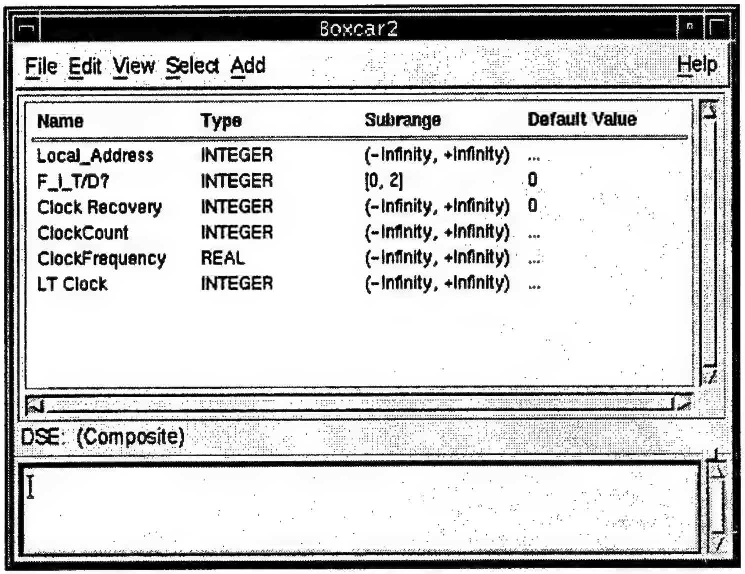


Figure 4.2.2 : Data Structure Editor Hierarchy

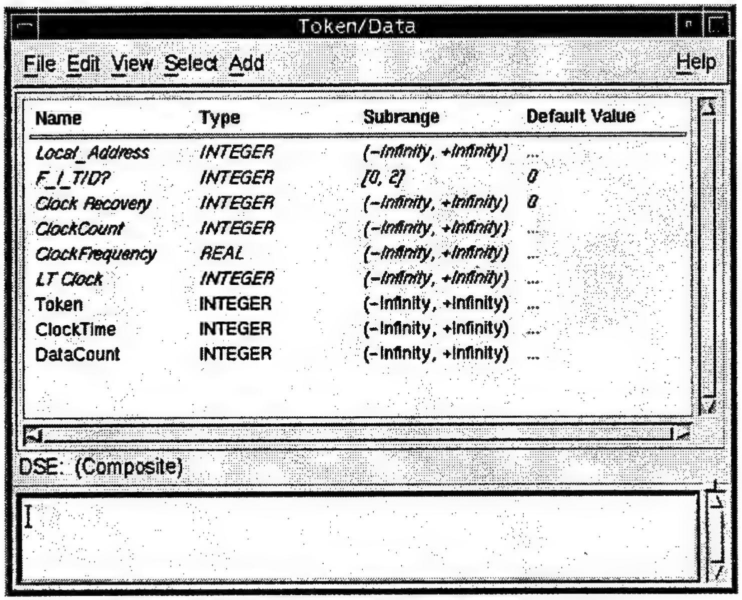
The BONeS data structure is graphically depicted and may be edited by double-clicking on the structure to edit. Each data structure holds fields, such as shown in Figure 4.2.3.

However, it may be more correct to think of aggregation working in the other direction. Children are composed of fields local to themselves and the fields contained in each super above them. This is shown below in Figure 4.2.3 and Figure 4.2.4. Notice that the field Token/Data inherits the fields of its super Boxcar2, just as a house is composed of its aggregate parts. Unfortunately, this does not map well to an aggregation chart unless the aggregate parts are assumed to be the fields within each structure defined. In an aggregation chart an object may be composed of independent subparts. In BONeS, however, *Enc_Cargo* is an aggregation of Token/Data, Boxcar2, Root-Object, and Trigger, which are all interrelated, not independent.



Name	Type	Subrange	Default Value
Local_Address	INTEGER	(-Infinity, +Infinity)	...
F_I_T/D?	INTEGER	[0, 2]	0
Clock Recovery	INTEGER	(-Infinity, +Infinity)	0
ClockCount	INTEGER	(-Infinity, +Infinity)	...
ClockFrequency	REAL	(-Infinity, +Infinity)	...
LT Clock	INTEGER	(-Infinity, +Infinity)	...

Figure 4.2.3 : Boxcar2 Data Structure Fields



Name	Type	Subrange	Default Value
Local_Address	*INTEGER*	*(-Infinity, +Infinity)*	...
F_I_T/D?	*INTEGER*	*[0, 2]*	*0*
Clock Recovery	*INTEGER*	*(-Infinity, +Infinity)*	*0*
ClockCount	*INTEGER*	*(-Infinity, +Infinity)*	...
ClockFrequency	*REAL*	*(-Infinity, +Infinity)*	...
LT Clock	*INTEGER*	*(-Infinity, +Infinity)*	...
Token	INTEGER	(-Infinity, +Infinity)	...
ClockTime	INTEGER	(-Infinity, +Infinity)	...
DataCount	INTEGER	(-Infinity, +Infinity)	...

Figure 4.2.4 : Inheritance of Data Structure Fields

The two figures above show the aggregation of fields into structures as the data tree branches out. Notice in Figure 4.2.4 that the *Token/Data* structure is composed of its own parts, *Token*, *ClockTime*, and *DataCount*, as well as the fields from *Boxcar2*: *Local Address, Clock Recovery, ClockCount,* etc., shown in Figure 4.2.3.

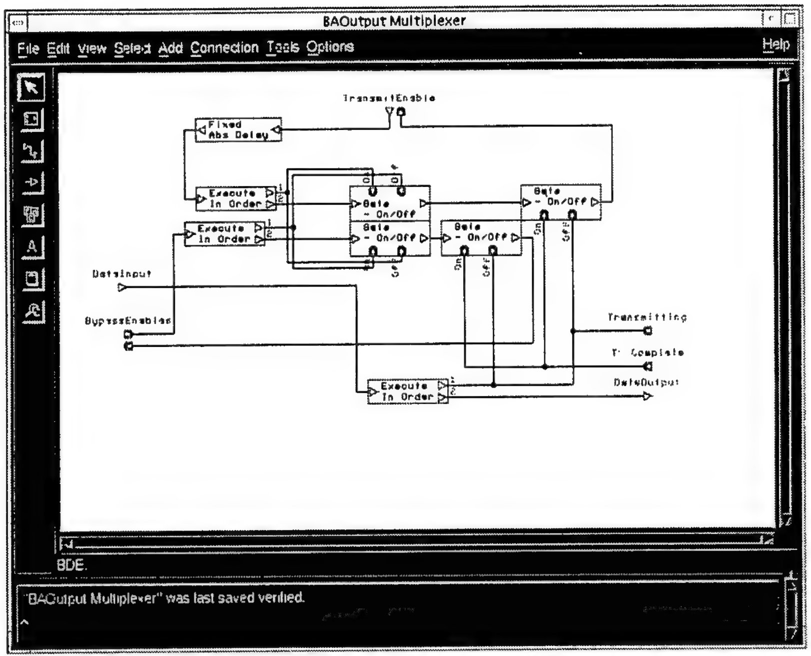
4.2.2 Libraries

The organizational structure of a BONeS model begins with the creation of a library. Every system, component, or probe must be associated with a library which is saved in the directory in which the library is created. However, in the BONeS group structure a component may be saved in any group no matter to which library it is associated. Models may also use components from other user libraries.

BONeS has built-in libraries which provide the foundation for most components the designer may need. The "Core Pool" libraries include components to create and manipulate packet structures, arithmetic operations, vector manipulation components, execution controls such as gates, delays, file manipulation, memory operations, queues, and many others.

4.2.3 Block Diagram Editor

A designer has many tools to create new models. Each component may be constructed with other components, ports, arguments, and connections. Any number of blocks may be added into a model and a hierarchical structure may be created with user-created components and core pool modules creating layers of abstraction at each new layer. Blocks communicate through ports. Each port must be designated a certain data type such as "Root-Object" and connections made to these ports must be either of that type or a subtype. In addition, generic blocks may be constructed using deferred ports. Deferred ports require that the data structure entering the port be a subtype (or of the same type) as the port. Deferred ports are useful in creating modules that may be recycled without modification such as a gate. Many core pool modules use deferred ports of the root-object type so that any data structure may enter and exit the module. BONeS refuses to make a connection if the subtype cannot enter the port. This offers a high-level form of debugging and is useful in detecting errors early. An example block diagram is shown below as Figure 4.2.5 taken from the bidirectional array.



"BACutput Multiplexer" was last saved verified.

Figure 4.2.5 : Output Multiplexer for the Bidirectional Array

The figure above is an example of the Block Diagram Editor in BONeS. The particular model is an output multiplexer of the bidirectional array.

Arguments are also a useful feature to implement into any component. Arguments allow the user to create structures which may be set to different qualities with the passing of parameters. For instance, the user may create a model with a FIFO structure and decide that he is unsure of the optimal queue size. This is also convenient for creating components which may be recycled later but could use different parameters. Parameters come in a few varieties including arguments and memory. Arguments are similar to those that are entered at the command line when executing programs. In fact, arguments function in exactly this manner. If set to a default value prior to compilation, the parameter is treated as a constant. Arguments may be internal or external. If they are internal, the argument must be given a default value. If they are external, the value of the parameter may be deferred to a higher layer. Parameters may be passed from layer to layer by linking a new parameter in the higher layer to a parameter in the lower layer. This allows the architect to pass or defer arguments all the way to the system model at the highest layer. An argument may also be a piece of memory. Memory, in this way, may be thought of as a variable parameter. A piece of memory may be used by many structures, not confined or integral to the block itself. Structures which use memory may link any data structure such as an integer or real which is useful in minimizing connections between blocks.

4.2.4 Symbol Editor

The Symbol Editor is a useful tool for pictorially representing block structures. It may be used to give a model a more intuitive graphical depiction. The symbol editor gives the user basic geometric shapes to construct symbols other than the common block. In addition, BONeS supplies a few frequently used shapes such as queues and other common network structures (e.g. WAN, LAN, Satellite Link, etc.). Any system or component may use a symbol instead of the default block.

4.2.5 Simulation Manager

The simulation manager is a very powerful feature of BONeS. Two important simulation modes supported are Background and Interactive. Interactive simulation mode allows the user to observe the data flow of the model down to the lowest layer. BONeS provides an array of tools to help the user debug any model such as tracing (step in) functions, breaks, and fast-forwarding. Any number of break point flags may be placed on ports to ensure proper data flow. In addition, the breaks may filter data so that only certain data events will trigger the break. Background simulation is used to receive feedback from the model more quickly. Although, the Interactive simulator writes to the probe files, Background simulating is much faster and allows another useful feature. Multiple iterations may be run concurrently or as a batch. The user has control to map a parameter to a range of values. The multiple iterations may be run on one machine or if available, multiple workstations. If run on multiple workstations, the simulations will run concurrently; otherwise, the simulations run as a batch.

4.2.6 Post Processor

The Post Processor is a two-dimensional graphing tool which may take any combination of input from files created from the probes after a simulation. Axes may be represented by a parameter, a probe file, "Tnow" (this is what BONeS calls the simulation time), or the sample number. Probe files may be filtered if the probe is very general and the user wants a more specific data set. For instance, a generic probe file may hold all of the traffic that entered a node's input port. However, the user may want to graph the number of idles which were input. The Post Processor may also present data in many formats (bar-graph, line, points) and more than one data set may occupy a single graph.

4.2.7 Primitive Editor

As mentioned previously, one of BONeS's most powerful features is the ability to create components in C/C++ code. Primitives may be constructed to build models which are unfeasible or impossible using the core primitives. Typically, a primitive is constructed by inserting ports into a BDE and then saving the file as a "primitive." The correlation between a primitive and C/C++ code is as follows: input ports are treated as variables, output ports typically are treated as an output function, and parameters are built as parameters. Using C++ primitives the designer may build the necessary functions needed to model most any network detail.

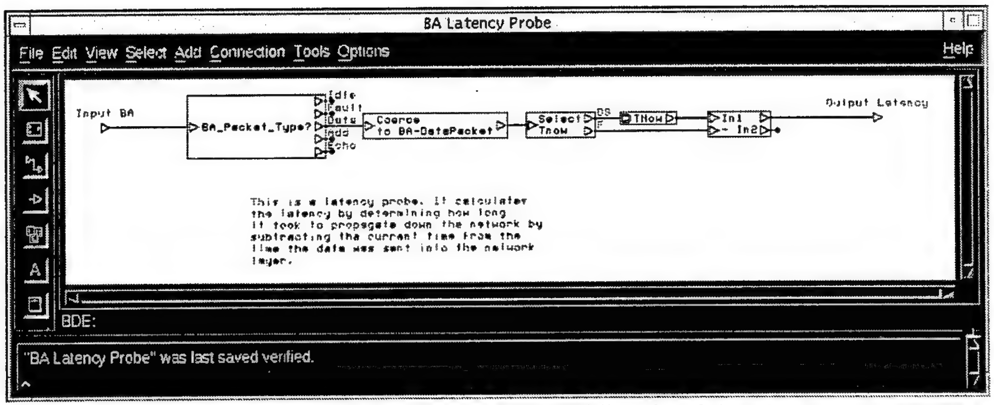
4.3 Performance Features

A probe is simply another block component with some limitations. Any component may be built into a probe just as a regular model is built, but only one input port and one output port are allowed. BONeS also provides generic probes of root-object type which may be set up to filter data types entering a node. To gather useful information requires some preprocessing. Creating a probe with block primitives and accessing data structure parameters provides more flexibility than the generic probes. Probes which are integrated into a model may record events in a file which may be plotted with the Post Processor. Three important probes were built into every DPSA model. These were latency, throughput, and utilization probes. Variations of these probes were also included in most models which take a running average of the latency and throughput results. This was done because the Post Processor lacks the tools to extrapolate statistical data. The formula for the running average is,

$$RunningAverage = \frac{PreviousRunningAverage * N + NewValue}{N+1}, \quad \text{(Eq.4.3.1.)}$$

where N is the number of values in the average previously calculated.

Latency probes are created in BONeS by time-stamping a packet before it exits the node. An extra field, usually called *Tnow*, was built into the data structures of each model to hold this value. When a packet enters a node (and more specifically the latency probe), the time stamp *Tnow* is stripped from the data structure and compared against the current time of the simulation. TNow (not to be confused with *Tnow*) is a simple core module integral to BONeS in the Number Generator Group which is triggered by any event and sends the current simulation time from its output port. Figure 4.3.1 shows the latency probe for the bidirectional array model.



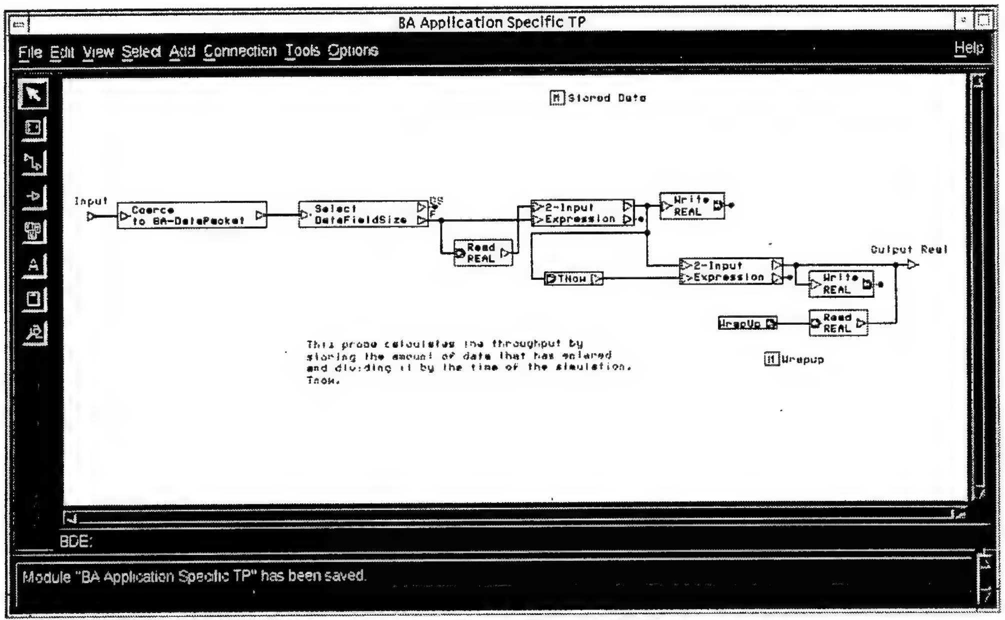
This is a latency probe. It calculates the latency by determining how long it took to propagate down the network by subtracting the current time from the time the data was sent into the network layer.

"BA Latency Probe" was last saved verified.

Figure 4.3.1 : Latency probe for Bidirectional Array

The latency probe simply pulls a field from the BA-DataPacket structure called Tnow and compares it against the current time of the simulation.

Throughput is more difficult to extract but any number of methods may be employed to extract this information. Three different methods were used to extract the throughput from the models. In one case, the packets were summed entering into the probe. Every time a data packet enters into the probe the instantaneous throughput is calculated by an algebraic equation which includes the number of packets other than data which arrived since the last data packet along with the network frequency. In short, we can observe the proportion of data bits to overhead bits and multiply this by the network frequency to find the instantaneous throughput. Another implementation of finding throughput was to count the number of data packets and divide by TNow. This is a natural way of continually finding the average throughput and is shown below as Figure 4.3.2.



This probe calculates the throughput by storing the amount of data that has entered and dividing it by the time of the simulation, Tnow.

Module "BA Application Specific TP" has been saved.

Figure 4.3.2 : Throughput probe for the Bidirectional Array

This probe calculates the throughput by dividing the number of bits of data entering the node by the simulation time. The WrapUp module will trigger the final average to be output at the end of a simulation.

Probes may be placed anywhere in the models. A latency probe would best be placed between the highest network layer and the application layer, whereas, a utilization probe may be useful on the physical link of the system at the lowest layer. Of course, this is a basic heuristic and these probes may be used effectively in many other locations. Appendix B demonstrates various other probes for gathering statistical data.

4.4 Fault-Injection Features

In order to test fault-tolerant features the models needed to include fault injection. Three faults that required critical attention were network failure, faulty nodes, and clock faults. Network and faulty node failures were handled in the same manner. Each simulated node contains a block through which all network inputs and outputs flow. This device can be programmed to sever connections to the network, thus rendering the node useless to the entire system and to the simulation. With a fault-tolerant optical bypass built in to the lowest layer, simulating a network failure would require programming two nodes to fail or to be removed from the network. Thus, the node downstream would sense a break in the network and re-initialize by becoming master and sending a fault packet. This fault packet informs each node to empty all buffers and reset as master. If only one node is shutdown from the network the following node may receive a signal from the preceding node. This is due to the optical bypass across each node.

A simple optical coupler could effectively bypass a bad node. This system has already been successfully implemented by Dr. Warren Rosen at the Navy Air Warfare Center- Aircraft Division (now with Drexel University). If a node fails, the following node may increase its sensitivity and use the information routed from the node behind the faulty one. This type of fault tolerance works on all of the topologies discussed above. Unfortunately, if two adjacent nodes were to fail, the system would be unlikely to recover. The opti-coupler, as shown below as Figure 4.4.1, is a simple -10dB passive bypass which means the signal amplitude after the bypass would be 10dB less than its original value. If two nodes were to fail the resultant signal to the next working node would be 20dB below the normal level.

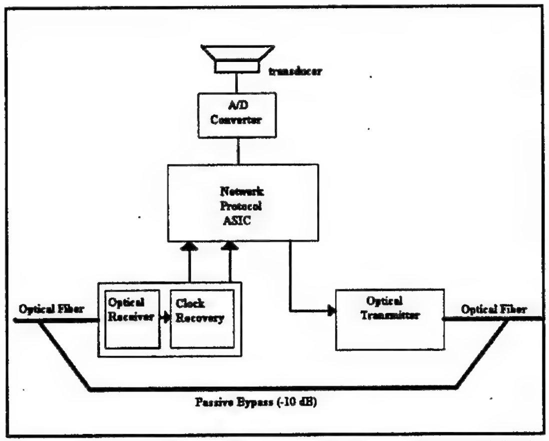


Figure 4.4.1 : Block Diagram of Sonar Array Node

The optical bypass shown in the figure above and implemented in the RDSA may also provide fault-tolerance for the parallel DPSA.

The resultant signal entering into each node is the main signal with the negative 10dB signal mixed with it. This additional mixed signal is usually interpreted as noise. An AGC (Automatic Gain Control) in the clock recovery device may increase its input sensitivity if the main signal is lost. Since it is impossible to implement signal hiding in BONeS, this system was modeled by two discrete inputs to every node. One input handled the main signal and the other the bypassed signal. The BONeS AGC circuit is shown in Appendix B.

The other fault-tolerance testing device built into the models was a skewing clock. This clock randomly wavers its output pulse times by a percentage of its average frequency. The pulse time is wavered by adding a Gaussian randomly fluctuating signal with a mean of zero and a standard deviation as a percentage of the average clock pulse width. This random signal generator is shown in Appendix B. Three parameters set the clock: the average clock pulse width, a percent skew error, and the low-pass filter coefficient. The clock has a hysteresis effect which is attributed to a second order low-pass filter (see Appendix B) so the clock wavers smoothly. The output of the device is shown below as Figure 4.4.2.

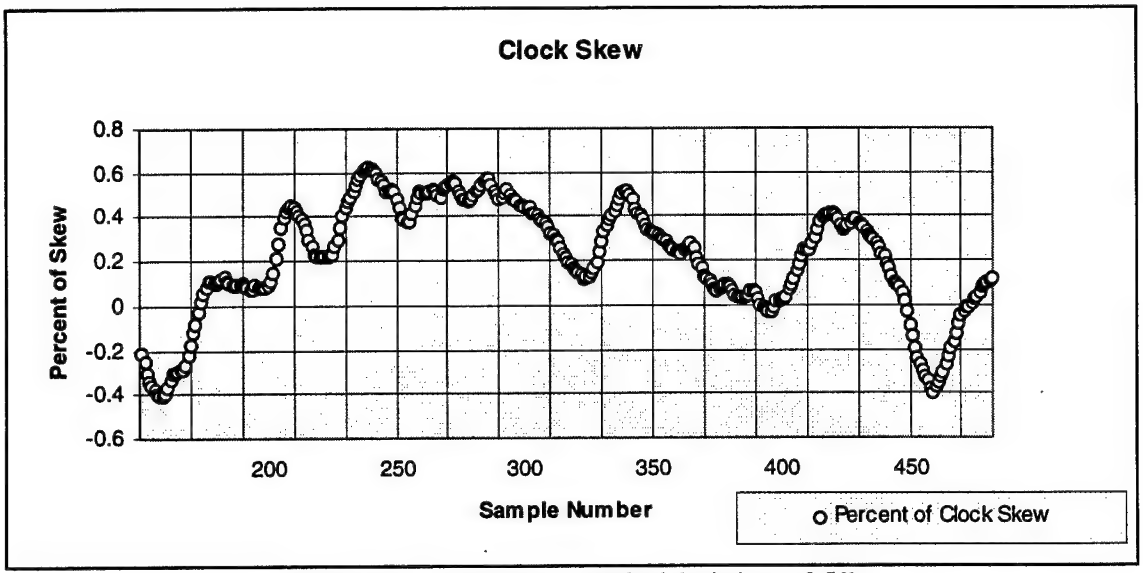


Figure 4.4.2 : Clock Skew, standard deviation at 0.5%

The wavering of the master clock is set with a standard deviation factor. In this plot this factor was set at 0.5%.

The filter is a basic second-order running-average filter, which filters the random signal generator. The coefficient is passed in as the parameter *LPcoefficient*. Another way of viewing this error is to plot the skewed clock pulse against the expected value of the clock. This is shown below as Figure 4.4.3 where the top points are the clock with skew and the bottom points are the expected or true values. The clock device is necessary in testing each network's tolerance to clock drifts and its ability to re-synchronize. It is also important in showing the network's ability to keep each transducer sample within reasonable synchronization and proper alignment. The clock was used in all the models tested.

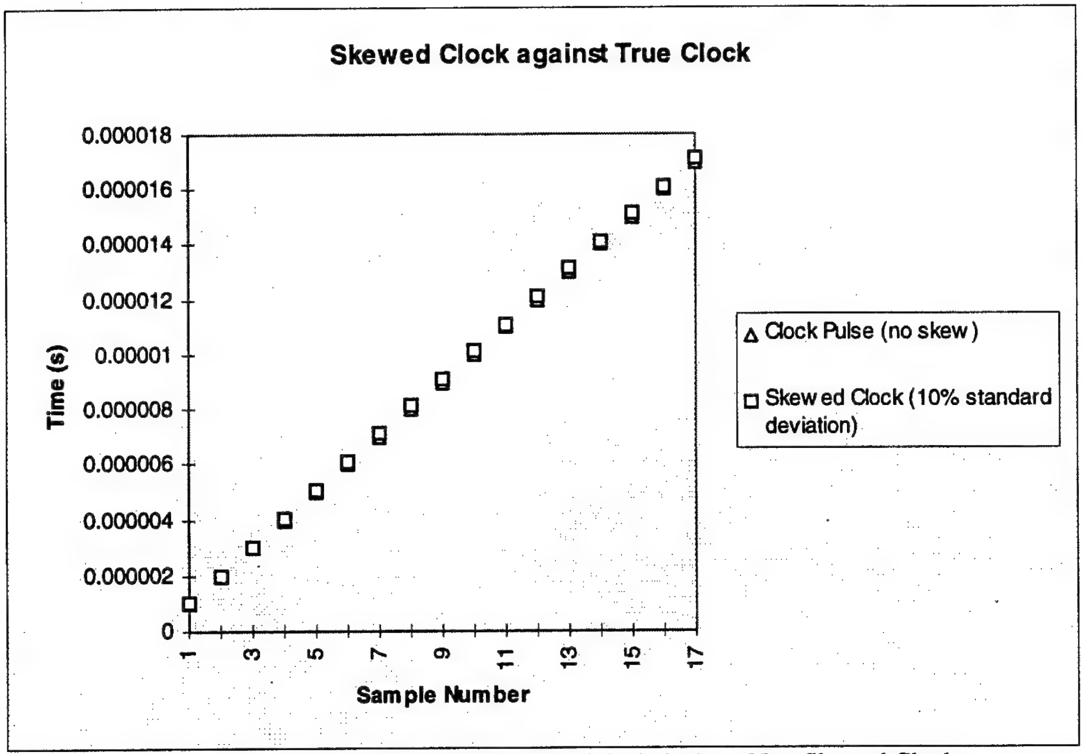


Figure 4.4.3 : Skewed Clock (10% standard deviation) vs. Non-Skewed Clock

The skewed clock shown above is slightly lagging the true value (the triangle behind it). The percentage of skew was unrealistically increased to 10% to accentuate the difference between signals.

4.5 Baseline Unidirectional Array

In the design of the BONeS model for the RDSA baseline, the protocol was simplified by many assumptions. The most obvious assumption, which has previously been discussed, is the assumption that no node will need to communicate with a node further upstream to itself. This implies that the furthest node will understand itself as the master node if there is no incoming data. Also, it was assumed that any node could become master for fault-tolerant measures. As master, the node is in charge of setting the network speed and creating the freight trains.

Another assumption made was that a destination field was unneeded. Each node was given a vector field within each freight train which would be sent to the processing node. Since each node is presumed to send to either the following node or the master node, no destination field is required and would present needless overhead.

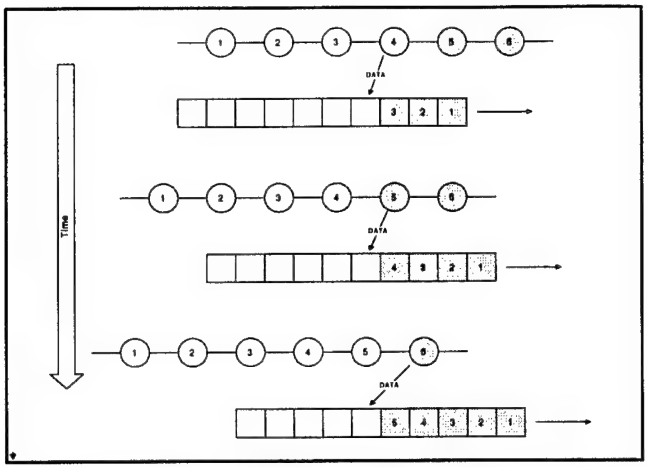
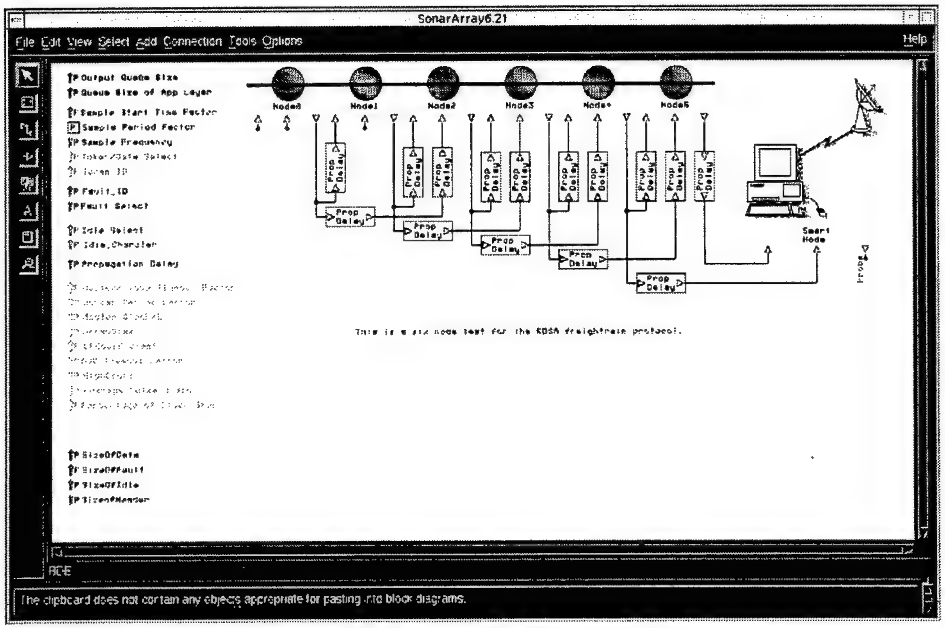


Figure 4.5.1 : Unidirectional Boxcar Protocol

The Boxcar2 protocol essentially reserves space for each upstream node to place its data into a "boxcar" of the passing train. Thus the data is emptied onto the train as it passes.

The unidirectional array, which is essentially the same network used as both the baseline and the parallel baseline, was modeled by six nodes. Its basic protocol passes a huge vector array freight train which is dependent on the size of the array. With a 6-node model the vector is small, but scaling the system up to a 120-node system also scales the size of the vector. Figure 4.3.2 shows the system model of the baseline architecture, named Boxcar2.



This is a six node test for the RDSA freighttrain protocol.

The clipboard does not contain any objects appropriate for pasting into block diagrams.

Figure 4.5.2 : Baseline Unidirectional Array

The system is composed of 6 dumb nodes and a processing node. The simulated processing node does not actually produce an output but injects proper latencies, etc.

The protocol is not store-and-forward but low-latency pipelined in order to lower the latency effects (shown below as Figure 4.5.3). This is simulated by accounting for time with delay units exiting the node and delay units entering into the node. The delays entering the node are small delays just to recognize the header of the packet and route it accordingly. The delays exiting the node require a more thorough explanation. The delays are not actually placed in line with the exiting packet. This is due to the nature of the feedback delay technique. The packet structure does not actually need to be delayed as it is sent out. Instead, what is more important is that the node is stalled from being able to transmit by placing a feedback delay before the node has control to send another packet. Since we are simulating data structures being passed around, not actually creating a model which sends binary signals out, the system is tricked into thinking part of the packet is arriving, and more of it is on the way, when in actual fact the simulated packets reach the destination instantaneously. This method, as referenced in this report, will be called *pipelined/feedback delay* and was used in every model.

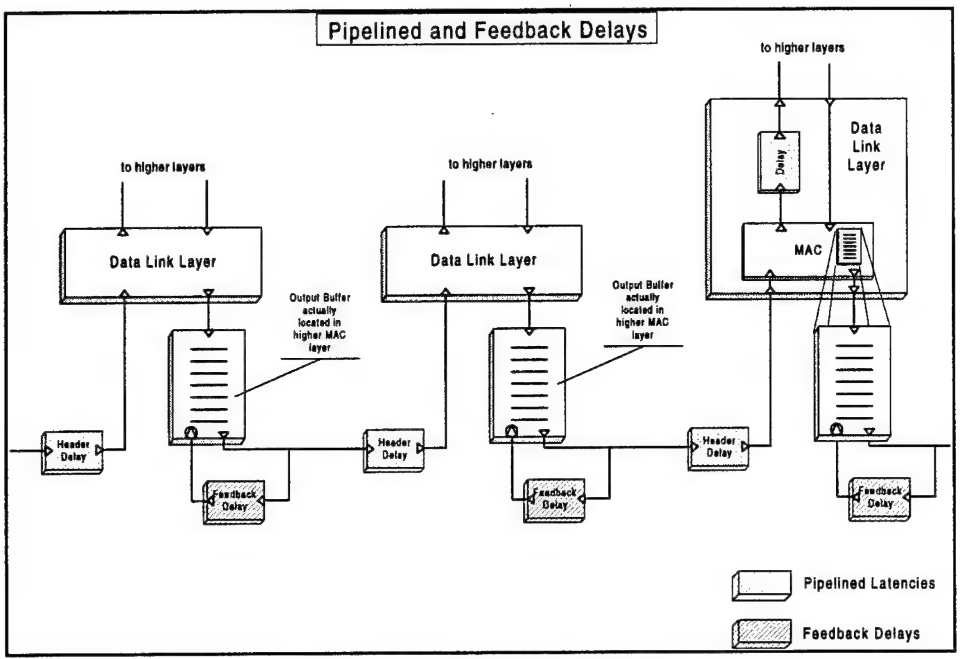


Figure 4.5.3 : Simulating Latencies

The pipelined latencies do not take into account the latency of the entire packet structure. Only when a packet has reached its destination is a packet delayed by its entire length.

The last critical delay which is required is the delay into the node which is actually receiving a signal. The final delay is placed in line with the packet before it is passed into the receiving node to preserve the correct network latencies. Thus the overall latency of the network in this situation would be the summation of all the header delays plus the latency to transmit the entire message plus the propagation times across each hop. Note that in boxcar this final delay is built into the smart node at the end of the network.

The application layer shown in Appendix B is composed of a clock divider circuit which cuts the clock time to the sample period. The traffic offered to the network is dependent on this variable and is set through an argument at simulation time. The baseline protocol is unique in that it does not have a true network buffer. Since there really is not an application, the sample buffer which also functions as the network buffer is in the application layer.

The Data Link Layer, which is typically composed of the MAC and the Logical Link Control (LLC), is shown Appendix B. Note that there is no DLL in the unidirectional array. This is because the DLL is used for acknowledged services; and since the unidirectional array cannot receive communication from the node to which it sends, it would be impossible for the receiving node to send back an acknowledge.

The MAC houses five main operations: the clock, the router, the master freight train generator, the data align mechanism, and the data packet append mechanisms. The clock is the same clock discussed previously and injects a certain amount of fault and uncertainty into the model. The router is a simple block which routes incoming data packets depending on their type. The master freight train generator creates a "boxcar" if there is a sample in the application layer buffer. The data align mechanism requires some explaining. Since the processing node at the head of the array assumes all of the data fields to be sampled at approximately the same time, the network must ensure that all the samples are aligned. This is done by the master node placing the sample time of the first packet into a special field. As the freight train passes, the node will place its data onto the vector array only if the samples are aligned. This mechanism has the ability to throw away old samples or place a zero in the field if the next data sample in the buffer is ahead in time compared with the samples on the freight train. The last set of blocks is used to place the piece of data into the freight train.

The physical layer, shown in Appendix B, holds the clock recovery device as well as the transmit and receive delays. The output is synchronized with the input clock frequency by the use of shared memory passed as a parameter. The master node or last node of the array sets the network frequency with its internal clock. All other nodes synchronize to this frequency by synching their output frequency to the incoming data stream. The output buffer, which is housed in the *SyncedOutput* module, sends an idle packet if no packet is ready for transmission and the queue is empty. Note that there is no bypass to the output buffer from the network input since the node assumes all incoming packets are destined for itself.

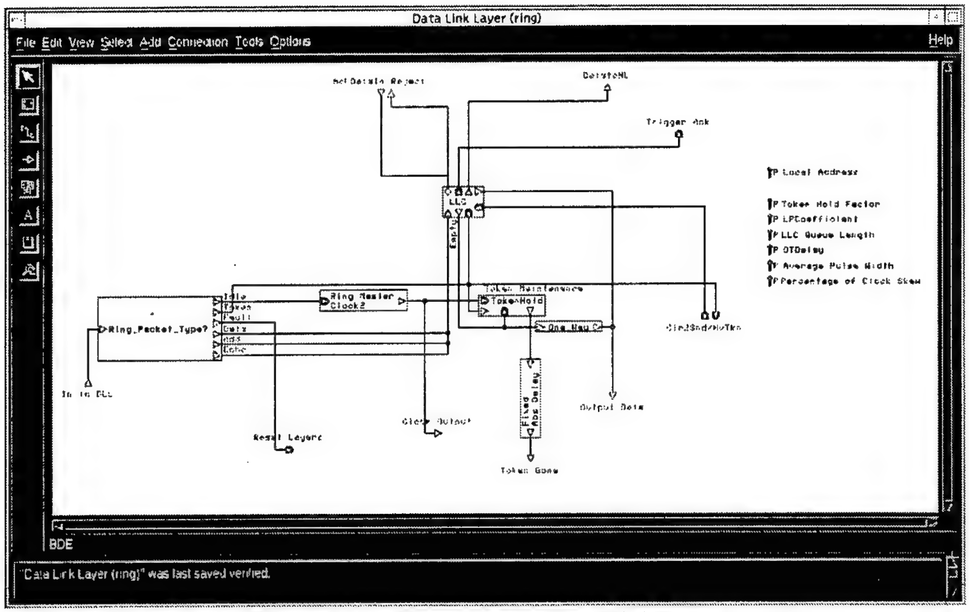
The RDSA freight train baseline has already been validated with a hardware model. The communication costs are great but the nodes do not include processing elements. The parallel implementation, although untested in hardware, should be feasible despite increased complexity of the nodes (particularly the addition of processing elements). This is assumed because of the reduction in communication through pipelined processing algorithms. The freight train protocol was composed of enough reserve elements for each node to load data. This enormous packet is sent down the array at each sample period, whereas the parallel implementation performs beamforming inline with the array so that by the time the much smaller packet arrives at the head of the array the result is computed. Using parallel processing elements and scalable algorithms, the device can be scalable to several hundred nodes while only affecting the latency of each computation. The throughput should remain the same no matter what the size of the array. The feasibility of the unidirectional array is a question of algorithm complexity. The array clearly cannot support robust algorithms which require acknowledged services, retransmission, or upstream transmission.

4.6 Token Ring

Not as many simplifying assumptions were possible for the token ring as were made for the baseline unidirectional array. This is particularly true since the ring is a fully-functional protocol. The first assumption, or rather decision, was to create a non-exhaustive MAC protocol. In other words, each node holds the token for a predetermined amount of time after which it must send out the token to the next node. This is different from an exhaustive protocol which sends data from the node until the output buffer is "exhausted" or emptied.

The token ring model created is composed of three layers: the application layer, the data link layer, and the physical layer. The application layer may use any number of traffic sources including a uniform generator, a Poisson source, a bursty generator, or a traffic source which uses deterministic data read from a file. This layer is also composed of a vector table which determines the addresses of live nodes. This may be useful in reconfiguring the network if a node is lost. In addition, it contains a FIFO structure which helps in containing traffic in bursty situations.

The DLL, shown below in Figure 4.6.1, is composed of both a MAC and an LLC. The LLC provides acknowledged service through the use of a wide FIFO. Each data packet that exits the LLC remains in the wide FIFO until the acknowledgment is returned. If the packet is not acknowledged after a circulation around the ring, the packet will be "peeked" again. This is modeled in the *TokenHold* block which holds the token for a certain number of clock counts (a number passed in as a parameter) or until the DLL buffer is exhausted. It also holds the master clock and the router which serve similar functions as explained in the baseline unidirectional case above.



"Data Link Layer (ring)" was last saved verified.

Figure 4.6.1 : Token Ring's Data Link Layer

The DLL of the token ring performs the duties of a MAC and holds the LLC sub-layer.

The physical layer is composed of input utilities, the input delays, and the output delays. The output is synchronized to the input if the node is not the master node. The master node performs utility functions such as setting the network speed and scrubbing the network for lost packets. The scrubber function is housed in the Input Utilities module along with the input header delay unit and the clock recovery unit. The output delay is not inline with the outgoing packets, just as in the unidirectional case. Instead, the delay is on the feedback path which allows the next packet to be sent.

The feasibility of such a protocol is determinate on the algorithm imposed on the network. Some algorithms seem to be very suitable for this architecture without saturating the network for cases of a small to medium network size. However, the biggest drawback with this architecture is its inability to scale to large systems successfully. The latency of the token passing to over 100 nodes would stall the network and thus result in low communication throughput. An alternative to this topology using the same network protocol would be ringlets, perhaps of 10 nodes in all, chained together with switches. The benefits of a system of this nature include a simple node protocol, much better fault-tolerance capabilities, and increased communication throughput. The disadvantage of this system would be an increase in hardware cost with the addition of 10 or 11 switches as shown below in Figure 4.6.2.

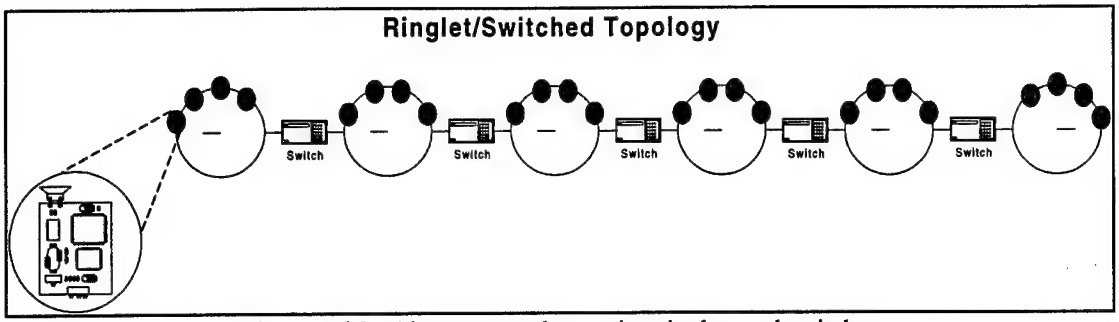
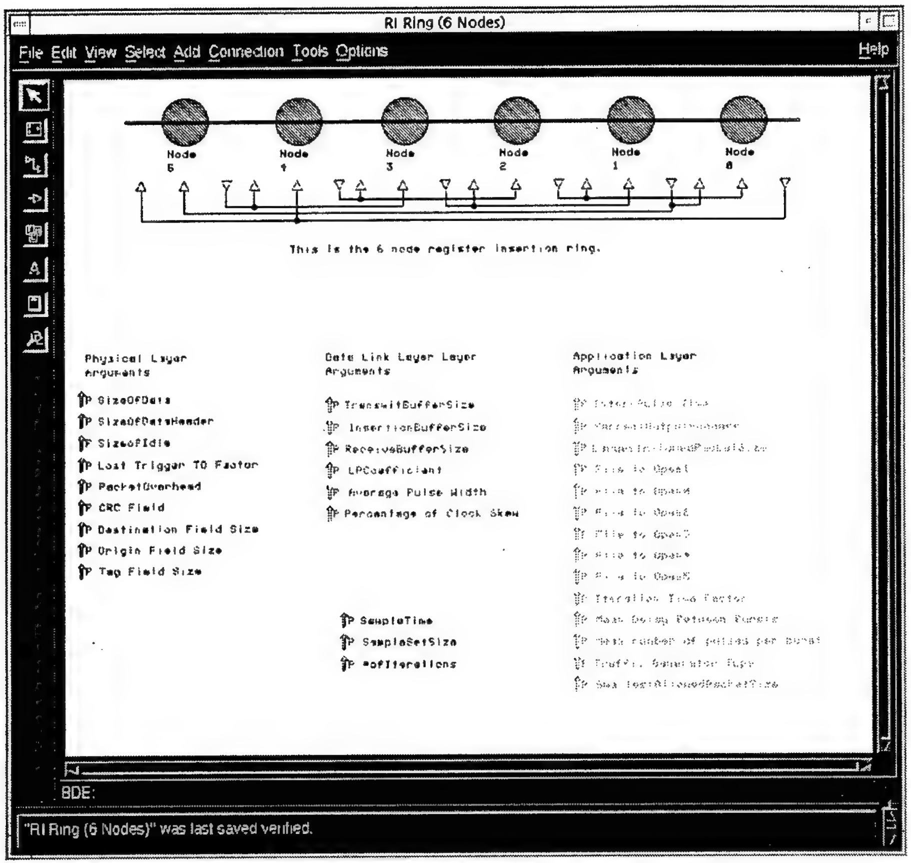


Figure 4.6.2 : Alternate topology using ringlets and switches

A ringlet/switched topology has more fault-tolerance than one large ring at the expense of increased complexity and extra hardware.

4.7 Insertion Ring

The insertion ring is very similar to the token ring with the exception of the MAC. The register insertion ring was found to have unacceptable blocking when random traffic was injected in the application layer. One way to correct this problem would be to use some sort of management protocol which would ensure fairness for each node. Another way to combat this problem would be to speed up the network to such a rate that no one node or combination of nodes could saturate the network. However, an assumption may be made here which simplifies the model and either allows the node to lower overhead by leaving out the management protocol or save power by clocking at a realistic rate. The traffic is expected to traverse the network in an absolutely deterministic way. In fact, we may inject the traffic patterns for any algorithm directly from a file. Because of this fact, we may presume that each node will offer approximately an equal amount of traffic. This result will also be used in the design of the bidirectional array. A BONeS block diagram of an insertion ring is shown in Figure 4.7.1.



This is the 6 node register insertion ring.

Physical Layer Arguments
- SizeOfData
- SizeOfDataHeader
- SizeofIdle
- Lost Trigger TO Factor
- PacketOverhead
- CRC Field
- Destination Field Size
- Origin Field Size
- Tag Field Size

Data Link Layer Layer Arguments
- TransmitBufferSize
- InsertionBufferSize
- ReceiveBufferSize
- LPCoefficient
- Average Pulse Width
- Percentage of Clock Skew

- SampleTime
- SampleSetSize
- #ofIterations

Application Layer Arguments
- Inter-Pulse Time
- [illegible]
- [illegible]
- File to Open1
- File to Open4
- File to Open6
- File to Open3
- File to Open2
- File to Open5
- Iteration Time Factor
- Mean Delay Between Bursts
- Mean number of pulses per burst
- Traffic Generator Type
- Max [illegible]AllowedPacketSize

"RI Ring (6 Nodes)" was last saved verified.

Figure 4.7.1 : Register Insertion System Model

This diagram shows the system level model of the insertion ring. The second input of each node is used to simulate the optical-bypass.

The application layer is simple but may be substituted with a number of different traffic generators. The generators constructed for this model are the same ones discussed for the token ring: a Poisson generator, a uniform generator, a bursty generator, and deterministic data read from a file. Note that in all of these models actual data is not injected into the data structures, but simply the size of the data structure or packet is recorded.

As stated before, the register insertion MAC is created with three buffers: the input buffer, the output buffer, and the insertion buffer. The output multiplexer switches the output path between the insertion buffer and the output buffer. Typically, the protocol only allows the output buffer to transmit when the insertion buffer is empty. However, in advanced insertion networks, priority modes and flow control complicate this basic operation. The input switch routes packets to the insertion buffer or the input buffer if the destination field matches that of the node. As mentioned previously, this implies that the receiving node pulls network traffic from the ring, which is unlike the token ring. Note that a delay unit is placed between the input switch and the input buffer. This delay is used to inject the proper latencies incurred by the transmission of the data. Recall that the delay is not inline with the data packet when it leaves a node. This is the case because it is necessary to extract information such as timing and synchronization from the data packet immediately. However, this delay needs to be added before the node receives the actual data packet.

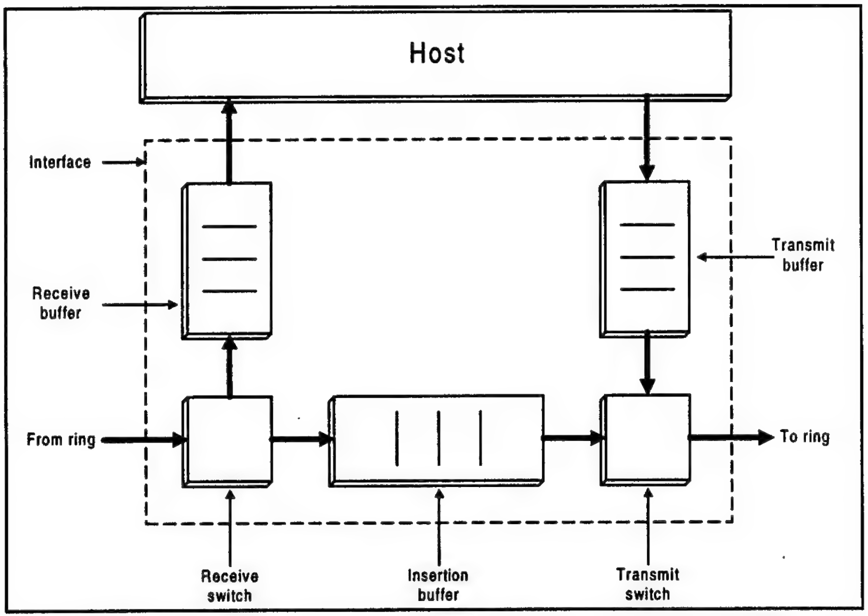


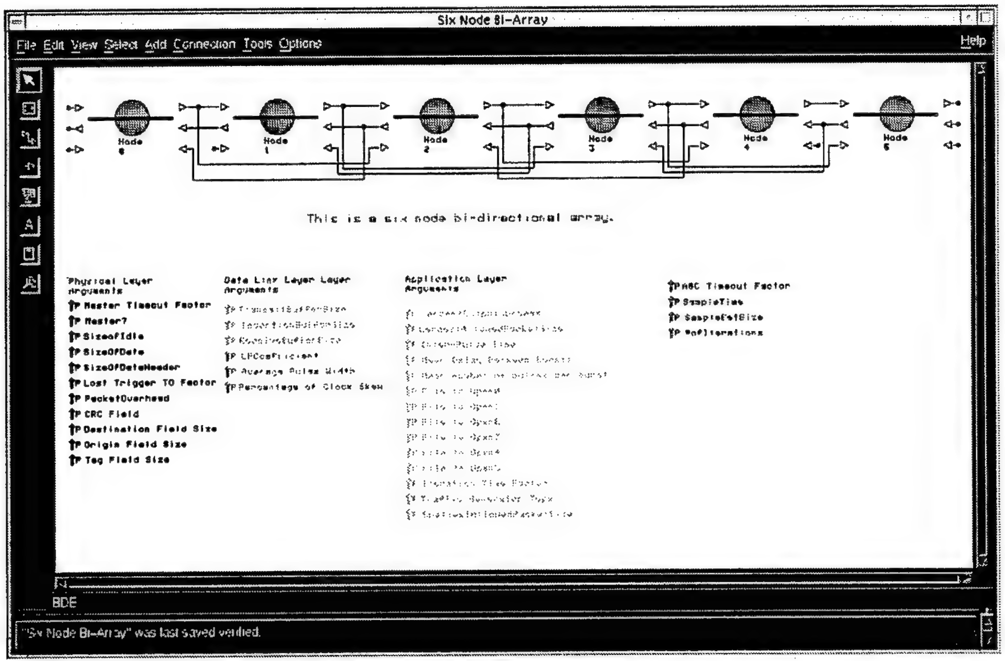
Figure 4.7.2 : Block Diagram of Insertion Ring Node [HAMM88]
The MAC layer of the insertion ring is composed of three buffers: the output, input, and insertion.

The physical layer houses three utilities: the incoming delay unit, the outgoing delay unit, and the clock synchronizer. Since no node is naturally the master node, either a random arbitration technique may be used to set the master or the nodes may be hardware set to be the master. The model sends a parameter to tell the node whether or not it is the master. If it is the master, its own clock sets the network speed. All other nodes will synchronize to this signal. Unfortunately this injects a single point of failure into the model but may be corrected by an arbitration technique for setting the network clock.

The insertion ring shows more promise as a functional ring topology. The theoretical maximum instantaneous throughput of the network is equal to the network rate, *R*, times the number of nodes, *N*, of the network. This is true for the determinate case in which each node sends data to its immediate neighbor downstream. The worst possible scenario would be if each node sent data to its immediate neighbor upstream. In this case, each data packet would have to traverse the entire hop count of the network and the total effective throughput would drop to the network rate *R*. For a 100-node ring, the total effective throughput rate for the best case compared to the worst case is an overwhelming loss. Assuming stochastic traffic, insertion rings are not known as a very scalable architecture, especially with over 100 nodes. Fortunately, the traffic which is intended for the network is not stochastic and may be designed in such a manner as to take advantage of the network's best feature, that is a theoretical peak network throughput of $N*R$. Careful algorithm design may prove that this architecture is a very viable system. Its obvious weakness, again, is its lack of fault tolerance which makes it difficult to improve without a revised topology such as the cascaded, switched-ringlet method discussed previously.

4.8 Bidirectional Linear Array

The assumptions for the bidirectional array are founded on similar principles as the register insertion ring. Since deterministic traffic will be on the network, the protocol has little need for management protocols which ensure fairness and congestion routing. One assumption about the network is that the end nodes do not need to reroute the packets back in the other direction. If a packet is destined towards a node in the opposite direction and never gets pulled from the network, the packet is simply sunk at the end. Because of this, the network layer will have to make a decision as to which direction to send a packet. This requires that each node knows where its destination node resides. This is a good assumption because sending data in the proper direction will also decrease the number of hop counts, thus resulting in a more efficient network. Figure 4.8.1 shows a block diagram of the system model of the bidirectional array.



This is a six node bi-directional array.

Physical Layer Arguments
- Master Timeout Factor
- Master?
- SizeofIdle
- SizeOfData
- SizeOfDataHeader
- Lost Trigger TO Factor
- PacketOverhead
- CRC Field
- Destination Field Size
- Origin Field Size
- Tag Field Size

Data Link Layer Layer Arguments
- [illegible]BufferSize
- InsertionBufferSize
- ReceiveBufferSize
- LPCoefficient
- Average Pulse Width
- Percentage of Clock Skew

Application Layer Arguments
- [illegible]
- [illegible]PacketSize
- [illegible]
- [illegible] Between Bursts
- [illegible] number of [illegible] per burst
- Bits to Send0
- Bits to Send1
- Bits to Send2
- Bits to Send3
- Bits to Send4
- Bits to Send5
- Iteration Time Factor
- [illegible]
- [illegible]PacketSize

- ABC Timeout Factor
- SampleTime
- SampleSetSize
- NofIterations

"Six Node Bi-Array" was last saved verified.

Figure 4.8.1 : System Diagram of Bidirectional Array

This model requires twice the number of input circuitry as any other model. However, besides increased complexity, the model scores well with performance and fault-tolerance.

The application layer is much like the register insertion model except, as of yet, no random traffic generators such as Poisson or bursty are integrated into the model. This is primarily the case because data from the beamform algorithms was collected before the model was constructed. Since a realistic traffic source was known, only the deterministic generators were included. Future tests will include random traffic generators for a study of the networks stochastic properties.

The network layer, as stated previously, is simply used to route the traffic in the correct direction. Each node is given an integer address of the counting set beginning with zero. The routing in a node is done by simply comparing its own value to the destination value and routing depending on the sign of the result. Another useful utility, although not built into the current model, is link-failure mechanisms. For fault-tolerant measures, the network has to carefully monitor the status of each node. Depending on the beamforming algorithm, the network may need to reinitialize to compensate for lost nodes. Some management protocol in the network layer could handle this task.

The data link control layer is similar to the DLL of the register insertion ring. The bidirectional array MAC protocol is based on an insertion buffer concept. The protocol requires two sets (or a symmetric pair) of many functions which illustrates one of the bidirectional array's greatest faults: an increase in hardware. These mechanisms include two outgoing multiplexers, two output buffers, and two input routers. This layer, as in all the other models, also houses the master clock.

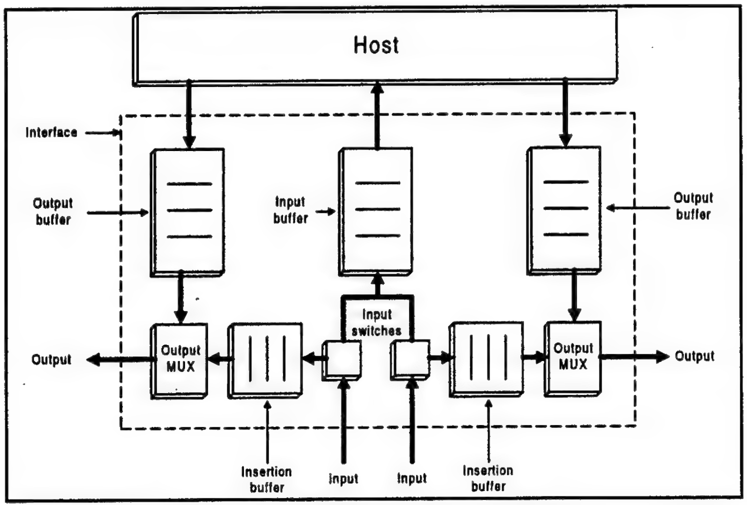


Figure 4.8.2 : Bidirectional Array Node

The MAC layer of the bidirectional array is essentially modeled as two insertion ring nodes. The two input streams share a common input buffer.

The physical layer, similar to the DLL, requires a symmetric pair of input and output ports and clock recovery blocks. The end nodes set the network clock rate. They are arbitrated by sampling the input signal. If no signal exists, then the node is assumed to be at the end of the array. Of course, the two end nodes set the clock speed on only one output. In other words, two clocks synchronize the array, one for the left transmitting traffic and one for the right.

The feasibility of such a protocol is a question of power constraints. It is certainly capable of handling traffic more efficiently than all of the other protocols and it is inherently the most fault-tolerant topology studied. The bidirectional topology provides a fabric for complicated communication patterns which, in turn, will support more complicated beamform algorithms. With the increase in the number of devices, the network protocol has about twice the complexity of the other models and power requirements will be at a premium.

4.9 Network Simulation Results

Two systems were used for each model, one for validation and the other for performance metrics. The validity models each used two nodes to generate the most basic data passing and to demonstrate the medium access control. The performance models were each six nodes in length and were used to gather throughput and latency statistics as well as utilization of the network topology.

The 2-node test was offered a uniform traffic pattern which is a standard traffic primitive in the BONeS suite. The traffic injected and other control packets were observed to verify each model. Deterministic tests were injected into the 6-node performance models by creating modules which could read from a file. The file format included both the destination address as well as the packet size in bytes derived from performance results taken from the Parallel Bidirectional Array FFT beamformer (PBF) and the Parallel Ring FFT beamformer (PRF). These preliminary algorithms, named according to their characteristics, will be discussed in detail in Chapter 5. The results of these tests were recompiled into a format useful for the BONeS file handler. The data was then parsed from the file by reading a pair of lines at a time; the first line was the destination and the second line was the size of the packet. PBF and PRF algorithms were run on 6 nodes at 64 samples per transform and run for 18 iterations. The number 18 is derived from the number of solution cycles (3 in this case) around the network as discussed in Chapter 5 (Algorithm Development and Modeling). The data was presented as the number of data bytes passed per iteration of the beamform solution. This restricts the network to communicate all of the data in the window of time needed to sample the set of data. The window was made more restrictive, however, by assuming that iterations overlapped sampled data sets. This overlapping factor was set to a medium case scenario in which data sets differed by approximately four samples. Figure 4.9.1 below shows an example of the overlapped sampled sets for three scenarios: large overlap, medium overlap, and no overlap.

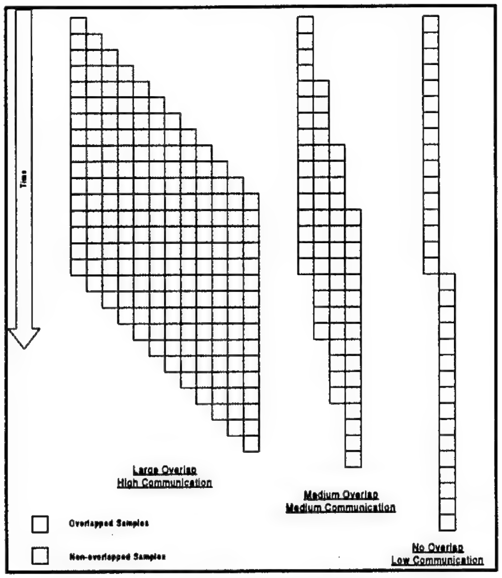


Figure 4.9.1 : Overlapping Sample Sets

The level of communication offered to the network is fundamentally based on the beamform algorithm, particularly the amount of overlap between sample sets

The offered load to the system is adjusted by the iteration time which is defined as:

$$IterationTime = \frac{N}{F_s(M+1)}, \qquad \text{(Eq. 4.9.1)}$$

where N is equal to the number of samples per iteration, M is the number of samples overlapped, and F_s is the sampling frequency. Since the overlap was restricted to the medium case scenario and the samples per iteration was fixed at 64, the only variable used to change offered load was F_s. The final algorithm may also adjust the samples overlapped if it is not sufficiently fast to handle the offered load. On the other hand, it may be possible to overlap the samples sets completely, in which case sample sets differ only by a single sample. Since the iterations were synchronized to the master clock device, they were triggered by a clock divider circuit which used the formula,

$$TriggerAfter = \lfloor IterationTime \cdot F_C \rfloor, \qquad \text{(Eq. 4.9.2)}$$

which is expressed in number of clock ticks, where F_C is the frequency of the master clock.

The parallel algorithms implemented for the array were medium-grained and took full advantage of the topology. PRF's data passing was for the most part sent to the following node downstream, and PBF's data was passed mostly to adjacent nodes. The statistics gathered from the parallel beamform algorithms still allowed much flexibility in actual implementation.

4.9.1 Results and Timings of Boxcar2

Boxcar2 was well suited to handle the traffic offered as demonstrated by Figure 4.9.2. The traffic injected into this model was deterministic but not handled in the same manner as the other network topologies. Since this is a model of the baseline protocol, no processing elements were assumed. Instead, the protocol sends the raw sampled data to the end processor.

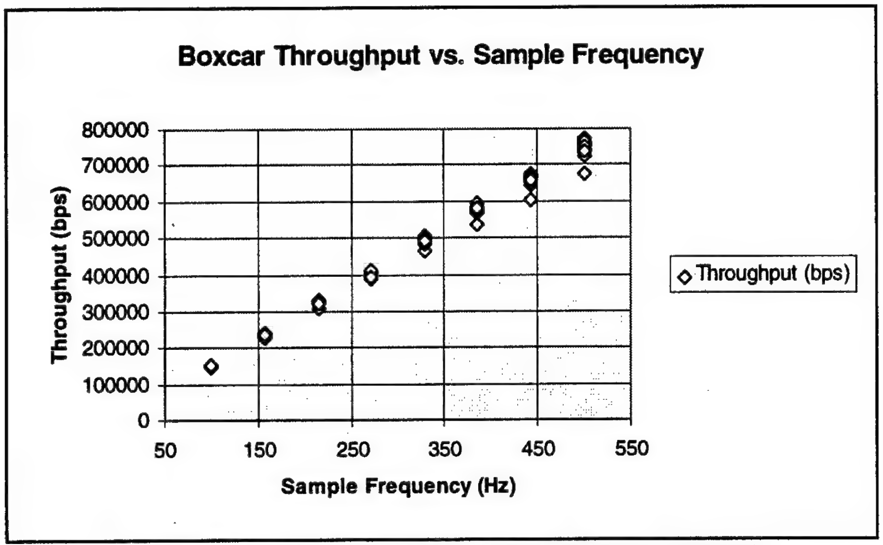


Figure 4.9.2 : Boxcar Throughput in Non-Saturated Network

Boxcar is lightly loaded at a network speed of 10MHz and 12 bits per sample. At realistic sampling rates and using the freight train protocol, the network is only 10% utilized. As a result, the baseline could have been clocked down at 1MHz.

The offered load is approximately equal to the number of nodes multiplied by the sampled data size which is assumed to be 16 bits after encoding 12-bit samples over one sample period. There is also some overhead for clock synchronization and control packets. The network was fixed at 10 Mbps, so again the only variable in changing offered load was the sample frequency. The throughput plot above shows the region between 100 and 500 Hz. Although not run out to saturation the simulations were run at realistic sampling rates. Since the network was far from saturation, the network speed may be clocked down, provided that no additional complexity is added to the protocol.

The saturation of the network is deterministic and unvarying. Since the size of the packet is constant and each packet follows the same path, each data packet should have approximately the same latency no matter what the offered load if small variations are allowed due to the skewing clock. Figure 4.9.3 verifies this result. The average latency for each node remains constant with sample frequency; the offered load does not have an effect on the latency.

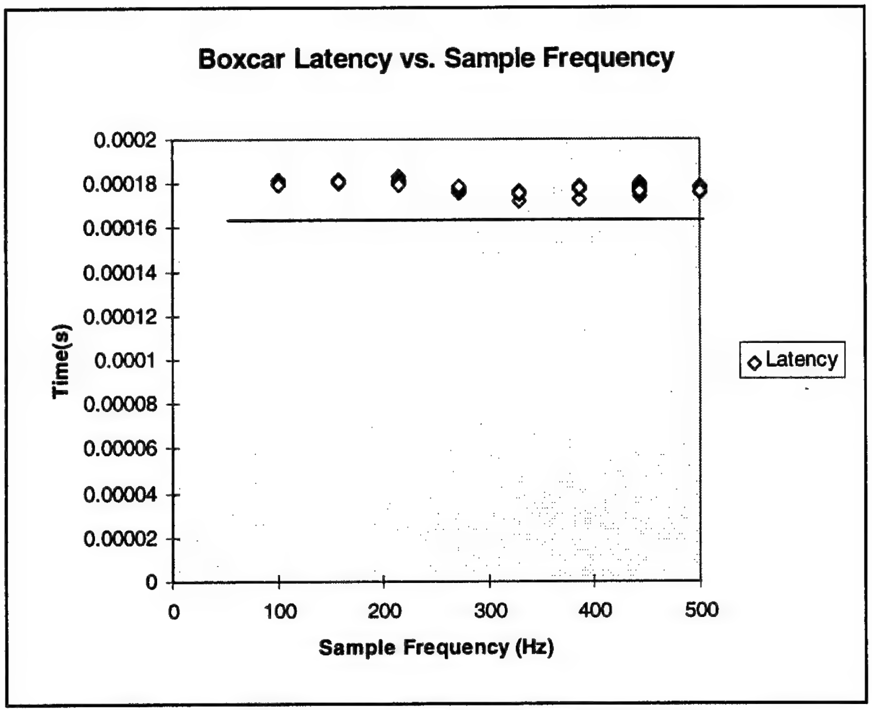


Figure 4.9.3 : Boxcar Latency vs. Sample Frequency

Boxcar's latency is directly related to the size of the network. The latency for this test is small although it is across six hops.

4.9.2 Results and Timings of Register Insertion Ring

The register insertion model was verified with a 2-node model and a maximum sampling frequency of 200 kHz. Figure 4.9.4 shows the average throughput versus the sampling frequency. Recall that the sampling frequency is based on a medium overlapped case in which sample sets are shifted by four samples. The saturation frequency for the injected program data, PRF, was approximately reached at 50 kHz. This high saturation point implies that the overlap may be decreased to a single sample, yet, still function at a maximum sampling rate of 12.5 kHz. An alternative to this would be to clock down the network speed, in effect saving power, to a quarter of the current rate of 10 Mbps. Decreasing the overlap could lower the network frequency another order of magnitude. At 75% overlap and a maximum sampling rate of 12.5 kHz the network could be clocked as slowly as 625 kHz assuming the PRF algorithm. With this much flexibility, register insertion is quite capable of handling the load of the network. Figure 4.9.5 shows the instantaneous throughputs at each frequency and for each node.

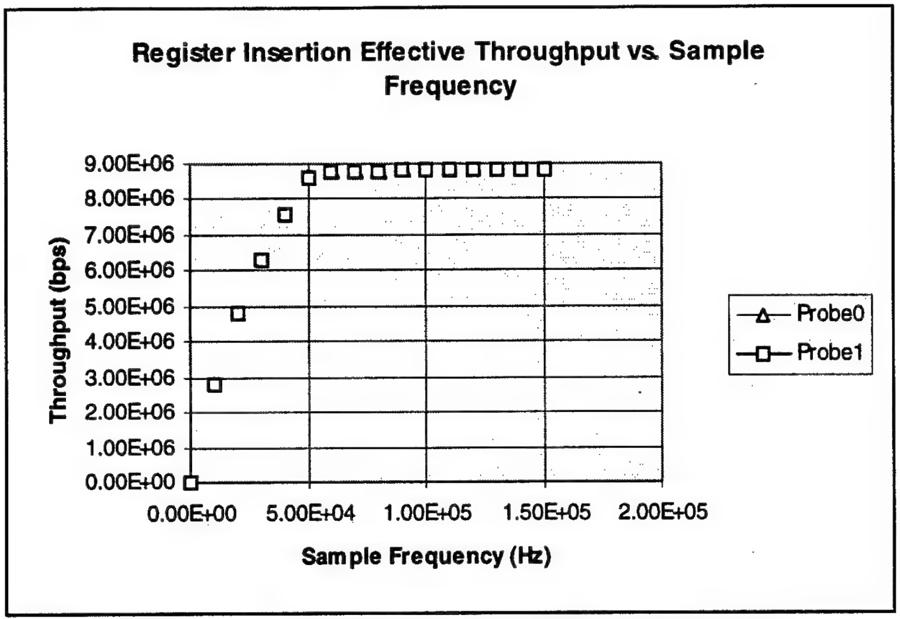


Figure 4.9.4 : Average Effective Throughput of Insertion Ring

This plot shows the saturation of average throughput for a 2-node test. The network speed was set at 10 MHz.

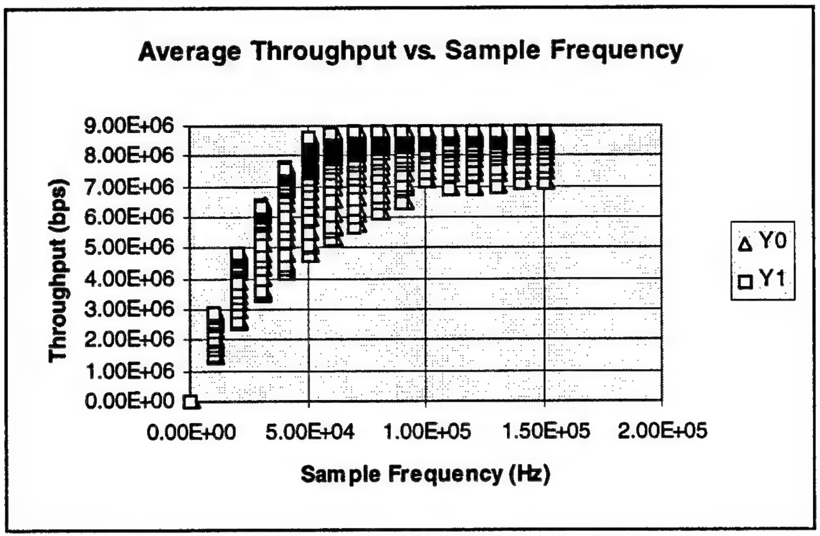


Figure 4.9.5 : Instantaneous Throughput of Insertion Ring

This plot shows instantaneous throughput for many packets in a 2-node test. The network saturates at about 50kHz. Y_0 is the effective throughput of node 0 and Y_1 is the effective throughput of node 1 of the 2-node simulation.

One drawback to relatively slow network speeds is the latency. Latencies were minimized by using the pipelined method as described above, yet with large packet structures, the latencies were on the order of milliseconds or tenths of a millisecond in the best case. On saturated networks the latencies ramp up to infinity, and packets spend more time in the insertion and output buffers than they do transmitting. Latency across one link is shown below in Figure 4.9.6. Furthermore, hops add at least the header delay divided by the network clock frequency under light conditions. Also note that the network latencies for the insertion ring are more spread out than the latencies of Boxcar.

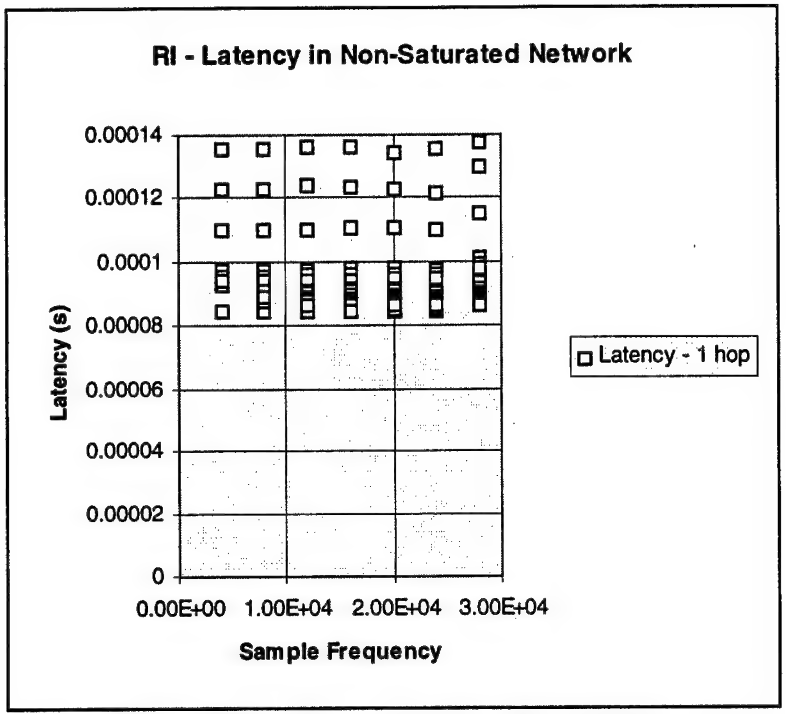


Figure 4.9.6 : Latency across 1 hop in Insertion Ring

This plot shows latencies for data samples within the non-saturation region. The latency of the network remains fairly constant on a lightly loaded network.

The PRF algorithm, run on a 6-node insertion ring, maintained high throughput and moderate loading at high sample frequencies. The network saturated at about 27 kHz as shown below in Figure 4.9.7. This plot shows the effective output throughput by node 0 of the insertion ring. Each node saturated at about the same level although some nodes offered more traffic to the network.

Maximum utilization of the network was reached at about 90% for the 6-node network. This is quite good when compared to commercially available networks such as Ethernet (CSMA/CD) which may have a utilization limit of 37% or less. Figure 4.9.8 shows this utilization limit when the offered load was ramped up to 40 kHz sampling frequency.

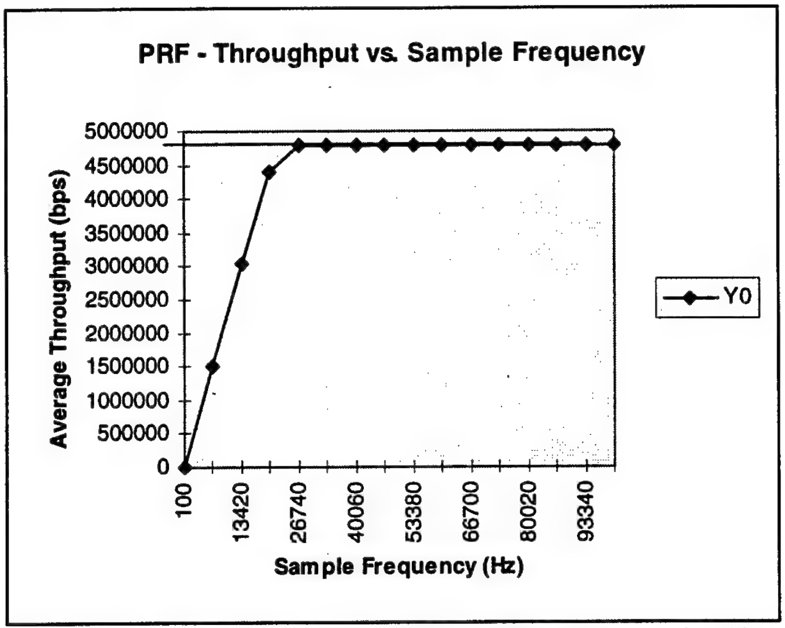


Figure 4.9.7 : Effective Throughput of PRF of Node 0 on Insertion Ring

PRF (Parallel Ring Frequency domain) beamform algorithm lowered the saturation sampling frequency to about 27 kHz. This plot shows the effective throughput of one node. Y_0 is the average throughput of node 0 in the 6-node simulation.

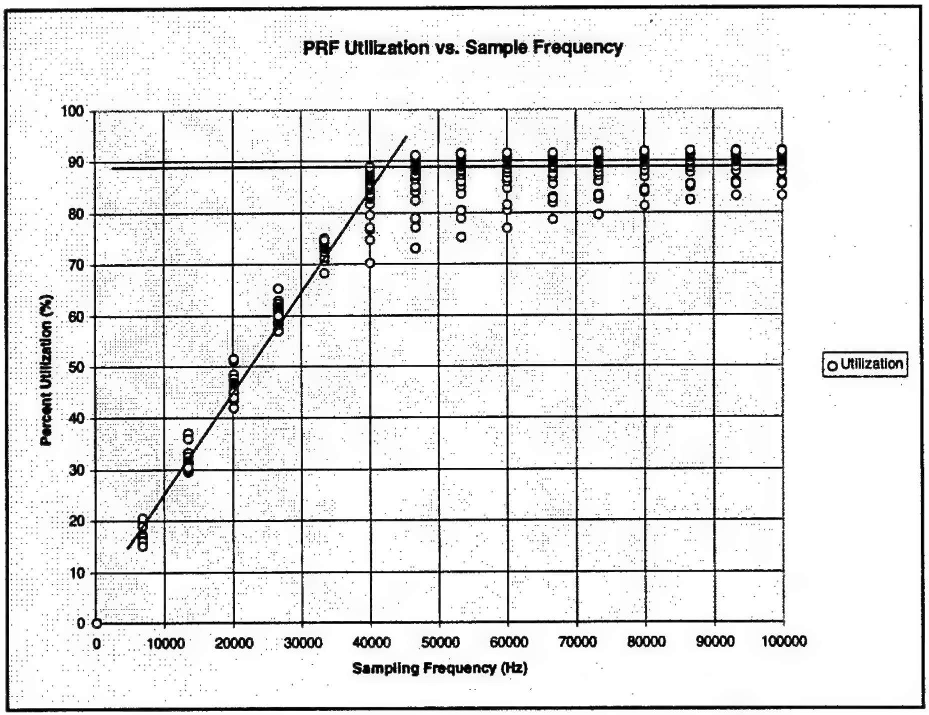


Figure 4.9.8 : Utilization of 6-Node Register Insertion Ring

The utilization of the network hits saturation at about 40kHz for the 6-node register insertion ring. The plot shows maximum utilization of 90%.

4.9.3 Results and Timings of Bidirectional Array

The results for the 2-node functional test of the bidirectional array were exactly as expected. The insertion ring and the bidirectional array were essentially created from the same philosophy: insertion buffers. Each MAC holds an insertion buffer, an output buffer and an input buffer. The bidirectional array has additional insertion and output buffers creating a symmetric pair. The protocol could have just as easily been created for a dual concentric register insertion ring. In the 2-node test, each node only used one half of its output capabilities, although, it could have been wired in a dual concentric ringlet fashion closing all connections and doubling the theoretical performance of the register insertion ringlet. However, since the network was not intended to be wired in such a fashion the outer ports were simply terminated. Thus, the bidirectional array acted exactly like the 2-node insertion ring. This is confirmed by the results shown below in Figure 4.9.9. The effective throughput of each node was approximately 9 Mbps and saturated by a sample frequency of 50 kHz.

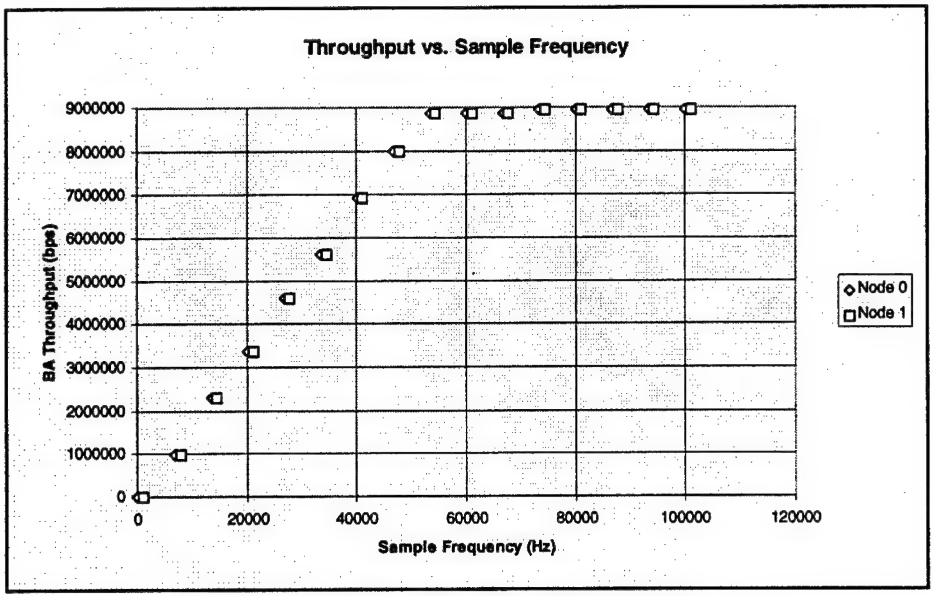


Figure 4.9.9 : Effective Throughput of Bidirectional Array

The 2-node functional test for the bidirectional array should show similar results when compared to the insertion ring. This is in fact the case. Both networks show a saturation frequency of 50 kHz.

The latencies of the bidirectional array within the operating region were similar to those shown previously with insertion ring. The latencies were on the order of tenths of a millisecond for a single hop. However, as the sample frequency was adjusted into the saturation region the latencies of the network are affected by time sitting in the insertion buffers or output buffers. Assuming an infinitely large output buffer size, the latencies of the network will ramp to infinity. Reaching this saturation point the latencies should immediately be disregarded since the network would no longer be able to support such traffic and latencies would then be considered infinite. This ramping effect is shown below in Figure 4.9.10.

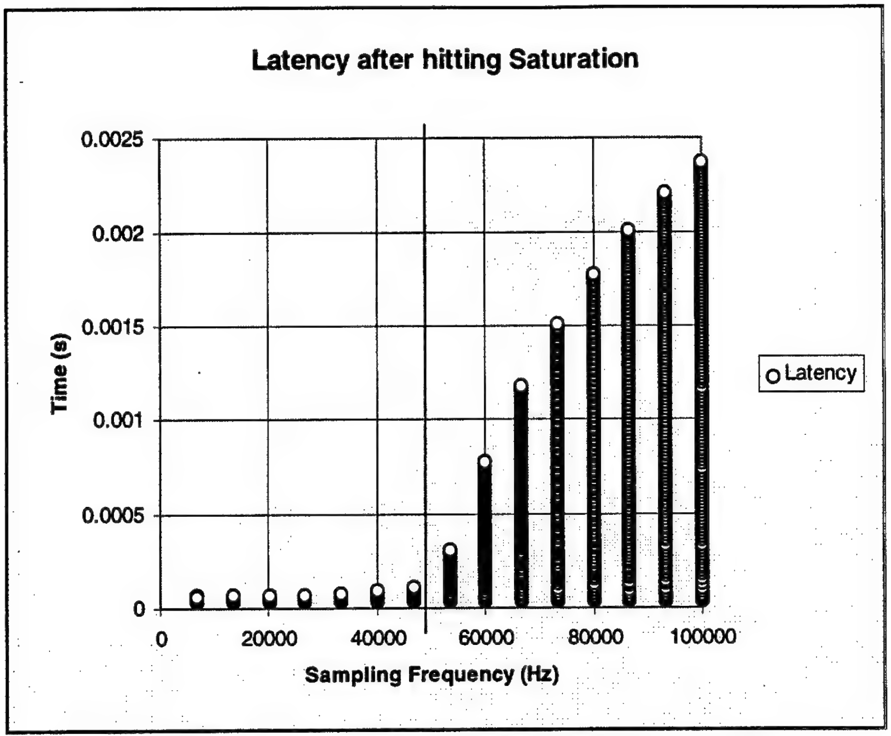


Figure 4.9.10 : Latency in 2-Node Bidirectional Array
This plot shows the latency of the bidirectional array network when operating in the saturation region. The vertical line in the middle of the chart delineates the saturation and normal operating conditions. The latency of the network will eventually ramp up to infinity as offered load is increased.

PBF was simulated on the 6-node model and the traffic was again adjusted by the sample frequency. The execution overlap was set to 4 as in the previous case. The differences between the insertion ring and the bidirectional array become apparent when observing Figure 4.9.11. The traffic offered by each node, although varying per node, allowed a maximum saturation of the network at about 40 kHz which is close to the theoretical limit imposed by the 2-node test of 50 kHz. This implies that either PBF was a more carefully constructed algorithm or that the bidirectional array is simply better suited to operate a parallel beamform array. In actual fact, either topology is suitable and capable of handling the type of traffic imposed by this system.

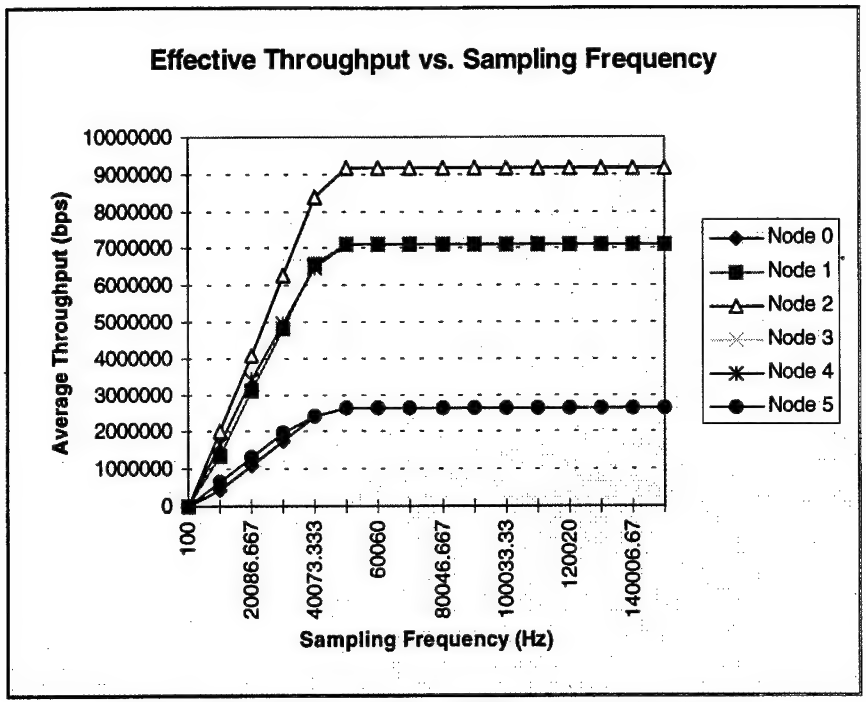


Figure 4.9.11 : Traffic Pattern of PBF on Bidirectional Array

The PBF (Parallel Bidirectional Frequency domain) beamform algorithm introduced the above network pattern on the 6-node model. The saturation point was lowered to about 40 kHz with six nodes.

4.9.4 The Ideal Topology

Each of the systems has thus far proved able to support the traffic offered to it, although the bidirectional array slightly outperformed the register insertion ring. The algorithms tested over these network models were of the most basic type, but the offered load at realistic sampling frequencies was barely enough to show that the networks were working. This implies that the bidirectional array and register insertion ring are clearly capable of handling robust beamform algorithms of many orders more complex or that the network clock speed may be run at low frequencies to save power. The token ring network, which is in the last stage of debugging, is expected to not fare as well as the other two protocols. Token passing is handicapped when there is an overly large number of nodes which must contend for a token. This does not rule out the possibility of a switched-ringlet network as mentioned previously. However, the additional complexity of such a scheme makes it unfeasible as compared to other alternatives such as the bidirectional array.

Performance aside, the bidirectional array, although somewhat more costly in terms of price and power, promises to be the best alternative for a parallel and distributed sonar array. It clearly handles traffic better than the other topologies besides having a natural advantage for fault-tolerance. Further, arbitration techniques for network utilities such as synchronizing the array simplifies the electronics and offers robust fault-tolerant measures.

5. Algorithm Development and Modeling

5.1 Generalized Message-Passing Framework

In order to choose an optimal beamforming algorithm for use on a distributed sonar array, models of the candidate algorithms were developed that simulated the dynamic computation and communication behaviors present in the algorithms. The first step in creating a model of an algorithm was the choice of programming paradigm. Since the nodes are physically distributed, a shared-memory implementation is costly and therefore unlikely. Hence, a shared-memory programming paradigm should not be chosen to accurately model the behavior of different algorithms.

Instead, a generalized message-passing model forms a natural fit to the structure of the sonar array. Any algorithm that can be simulated on a generalized message-passing fabric can also be executed on the distributed sonar array with no logical changes. To this end, a conceptual model of the entire sonar array system was developed that makes the transition from parallel programming paradigms to actual implementation a deterministic procedure. The abstract sonar array system model has three basic components: input sensors, processing nodes, and a message-passing network.

At this level of abstraction, specifics of implementation can be ignored and platform-independent algorithms can be developed. Platform independence is an important goal so that code developed for the simulator can be reused during the prototyping and final programming of the actual array. The conceptual model allows fast algorithm prototyping and debugging. The number of nodes in the array is completely arbitrary, as is the actual topology. The only restriction imposed on the model is the use of message-passing primitives as the sole method of inter-node communication. The only available commands for message passing are send and receive. All sends are guaranteed in that they are error free, immune to congestion, and always queued for reception at the destination node. Receives can be blocking or non-blocking. For a blocking receive, the node that executes the receive does not continue execution until the message it is waiting for arrives.

The only distinguishing feature of a particular node is its identification number which is an integer which begins at 0 for the first node and ends at *n*-1 for *n* nodes in a system. Messages are addressed to their destination based on the node numbers. Messages can be sent from any node to any other node without concern for routing or congestion through the network. Likewise, input from the sensors is automatically available to the processors through dedicated buffers, so that the programmer does not need to be concerned about taking data samples. Additionally, the sensor input is used for synchronization so that all nodes start their iterations at the same time. With this type of conceptual design model, the parallelization of beamforming algorithms on distributed sonar arrays becomes a message-passing decomposition problem, quite similar to the parallelization of other signal processing algorithms on a message-passing system.

This general framework effectively models a fully-connected, bounded-degree network (BDN). This type of network allows any node to communicate to any other node with messages routed through intermediate nodes. Nodes can therefore maintain a smaller, bounded number of physical connections without restricting the total number of nodes in the system. Hence, the network allows arbitrary communications (logically fully-connected) using a finite number of physical connections per node (node degree). One topology found in existing sonar arrays is the linear, unidirectional network. This allows communication in only one direction and hence is not modeled by a BDN because communications cannot be arbitrary. The unidirectional linear array is better classified as a one-dimensional systolic array because the communication patterns are set solely by the network connections, not by the executing algorithm. For this reason, special algorithms are developed for this type of architecture. Figure 5.1.1 depicts the sonar system model.

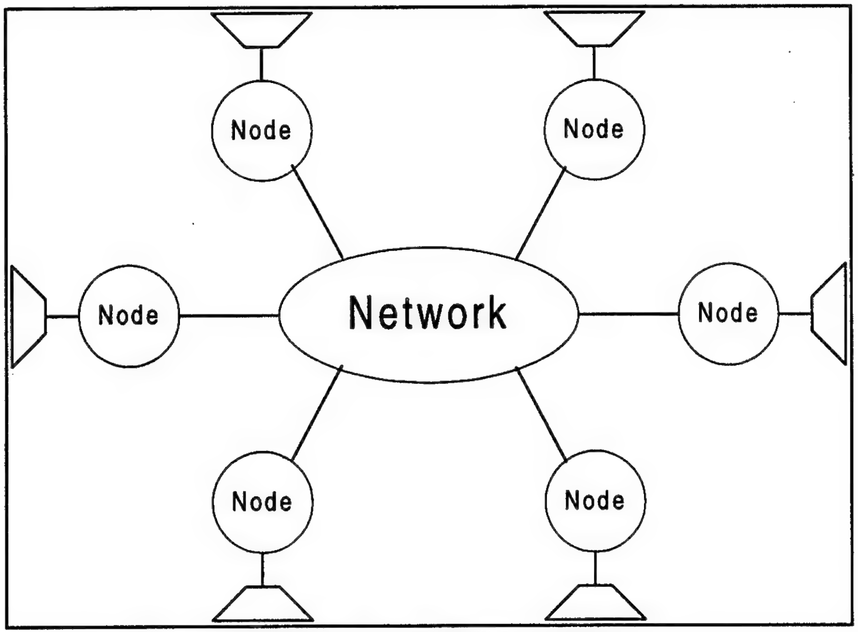


Figure 5.1.1 : Sonar System Model

This figure shows a representation of the BDN model which allows all nodes to talk to all other nodes through a network. The network is of no concern in the algorithm decomposition phase.

5.2 Beamforming Algorithms

Several representative beamforming algorithms from the time domain and the frequency domain were chosen to study for a number of reasons. These reasons include studying the feasibility of these algorithms in an implementation of a distributed sonar array, comparing the effectiveness of parallel decomposition of the algorithms, and establishing appropriate simulation mechanisms.

This section describes the algorithm development process using the generalized message-passing model. To develop parallel algorithms, the system abstraction is used to plan the decomposition. For example, it is assumed that a message-passing mechanism is used between all nodes, which allows for sends, receives, and a limited number of multiple-node communication schemes. Also, it is assumed that array nodes are their own entities and can easily be managed without the need to redesign the algorithm. Without needing to know the specifics of how an implementation (simulated or actual) provides these mechanisms, the researcher can design the parallel algorithms using the generalized model that mimics a sonar array.

To more effectively cover the several topics in this area, research into decomposition techniques and simulation techniques, such as multithreading and the abstracted framework, was done concurrently with separate algorithms. In the time-domain, the algorithms were developed by implementing the best of several well-researched decomposition methods, while complex multithreaded simulation was set aside. Heavy emphasis was placed in decomposing an iteration of the algorithm. Work in the frequency domain was accompanied by heavy development of appropriate simulation mechanisms. For example, the time-domain algorithms were simulated with as many array nodes as there are processors in the laboratory testbeds. The frequency-domain work was done by using several well-developed simulation mechanisms to run up to 128 array nodes in a simulated 128-node network on the same testbed which has much less than 128 processors. Additionally, work in the frequency domain involved the pipelining of multiple iterations of the algorithms.

Splitting the work into the above groupings is possible because of the nature of the algorithms developed thus far. Since the algorithms chosen were representative of basic beamforming, such as delay-and-sum and FFT, the computational requirements are not so demanding as to become a major concern in the parallelizing process. More attention can be paid to the simulation method or decomposition when the algorithm implementation is simple. Furthermore, since computation is small in the selected algorithms, the communication overhead costs become considerably more important; that is, the algorithms are inherently fairly fine-grained. Establishing efficient parallelism on such an algorithm is, as it should be, the bulk of the work.

5.3 Fault Resiliency of Beamforming Algorithms

An analysis of the effects of random node failures on the beam power pattern has been completed. The biggest effect of node failure is in the side lobe pattern rather than in the main beam. Figure 5.3.1 shows the beam power pattern for a 64-node array with half-wavelength spacing with no failed nodes. Figure 5.3.2 shows the pattern of the same array with 8 randomly failed nodes. The side lobe pattern is strongly dependent on the location of the failed nodes; however, the loss in the main beam power is independent of their location and depends only on the number of failed nodes. Figure 5.3.3 shows the loss in main beam power versus the number of failed nodes for a 64-node array. From this figure it is apparent that a substantial number of nodes may fail before the main beam power is significantly degraded. Therefore, it is essential to be able to bypass one or more failed nodes since the validity of the result gracefully degrades as the number of operational nodes is reduced. In addition, the algorithms may not need special fault-tolerant features to recover from faulty nodes since the result does not degrade quickly as the number of operational nodes decreases, but of course the architecture has to be capable of continued computation and communication despite hardware faults.

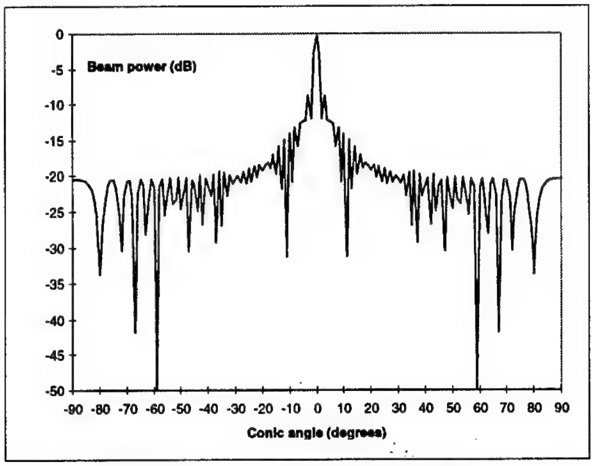


Figure 5.3.1 : Correct Beam Pattern

This figure shows the beam pattern for an array of 64 sensors with half-wavelength spacing with no failed nodes.

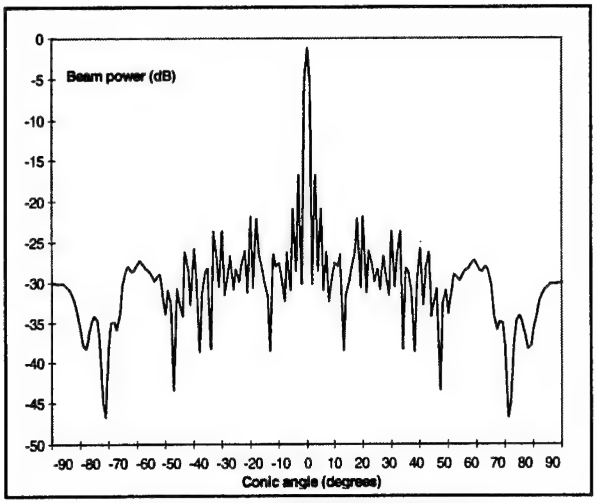


Figure 5.3.2 : Failed Beam Pattern

This figure shows the beam pattern for the same array with 8 randomly failed nodes.

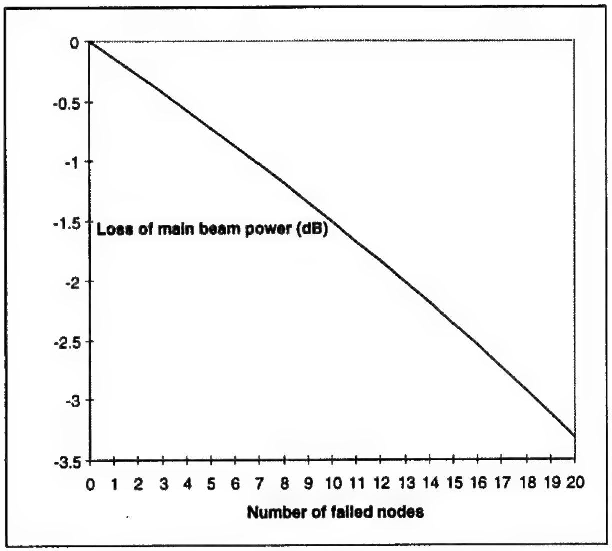


Figure 5.3.3 : Main Beam Power Loss

This figure shows the loss in the main beam power versus number of failed nodes for a 64-node array.

In an effort to further study the hypothesis that the effects of some failing nodes may not necessarily affect the final result to an extreme, assume that some mechanism in the network is able to automatically route around failed nodes. With this type of fault-tolerant network, the probability that the entire network will fail if the number of successive failed nodes exceeds the number that may be bypassed may be determined from simple combinatorial logic. A chart showing these probabilities is shown Figure 5.3.4. Clearly, the integrity of the algorithm is maintained during node failures and the probability of total network failure is very small with simple bypass capability in the network.

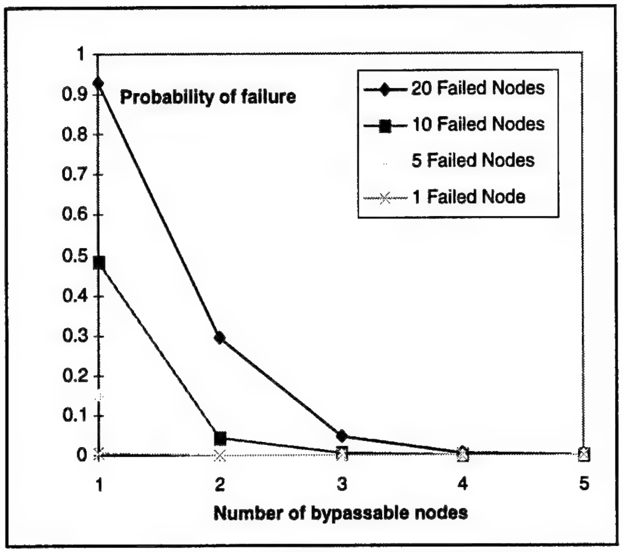


Figure 5.3.4. Probability of Network Failure

The graph above depicts the probability of network failure in the event of randomly failed nodes versus number of successive failed nodes which may be bypassed for an 112-node array.

5.4 Time-Domain Decomposition

The first sub-category in beamforming algorithms researched was the class of basic time-domain solutions. These algorithms are characterized by beamforming using time delays and without using spectral characteristics of the input waveform to determine beam power. Instead, techniques such as delay-and-sum and autocorrelation are used to form the beam.

5.4.1 Sequential Algorithm

The first time-domain algorithm explored is called delay-and-sum with interpolation (DSI). The second algorithm is called the autocorrelation algorithm. Both of these techniques were mathematically described in the Beamforming Theory background section. Due to the near equality of the two algorithms, the DSI algorithm is decomposed in the same manner that the ACC algorithm is decomposed, hence only the DSI decomposition is shown here. The method used in the following algorithms is to prepare the beam power pattern for several steering directions. This is computationally equivalent to detecting sources across the search region for several steered beams. A block diagram of the sequential beam pattern algorithm is given in Figure 5.4.1. Each block is labeled with a number for future references to the diagram. The first step in the sequential DSI is to initialize the array and incoming signal specifications. This includes the number of nodes in the array, the interpolation factor, node separation, speed of propagation, incoming signal frequency, sampling frequency, number of samples for each calculation, and the interval over which the beam pattern will span. Second, the number of steering directions must be calculated based on the improved resolution offered by the interpolation factor.

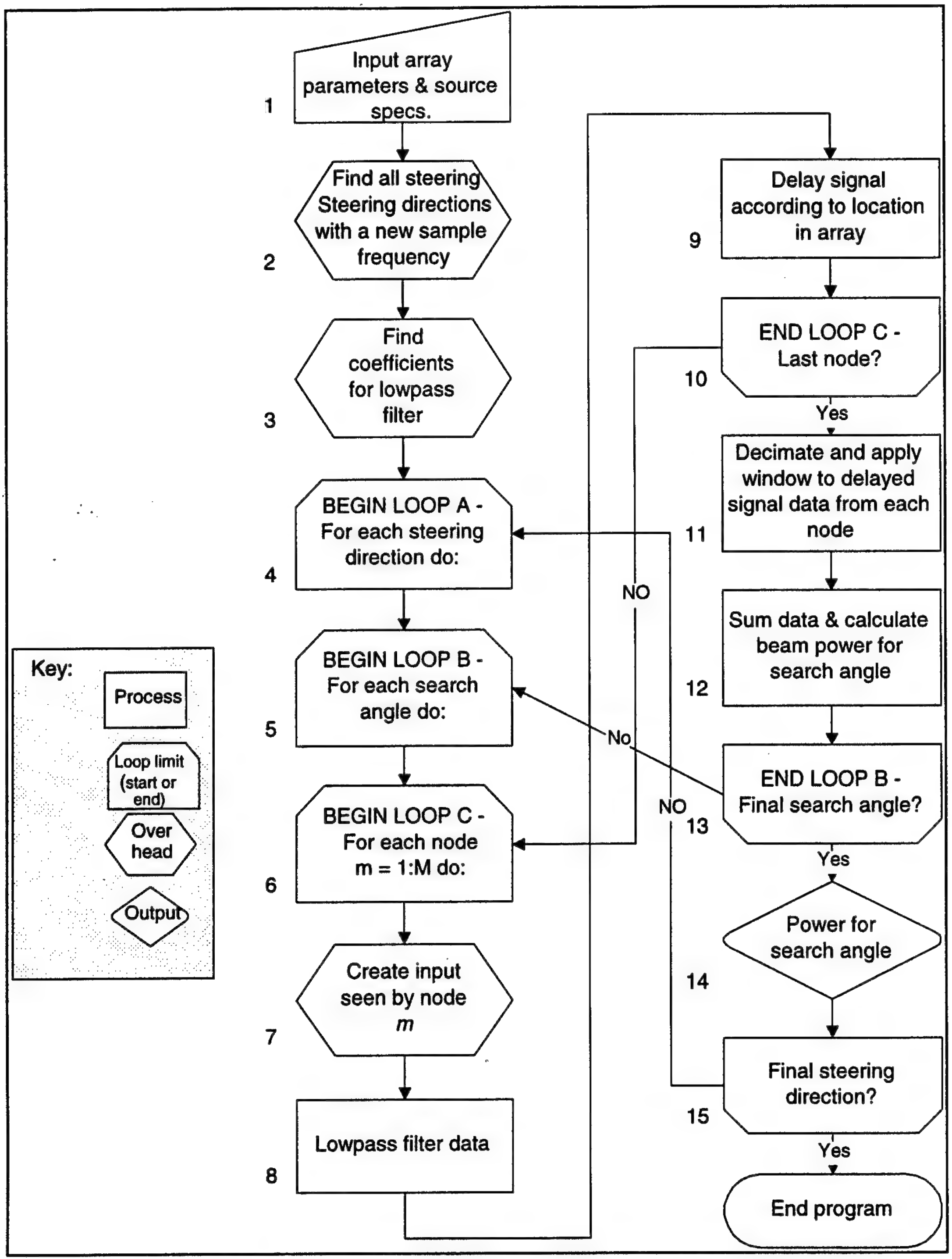

Input array parameters & source specs.
1
Find all steering Steering directions with a new sample frequency
2
Find coefficients for lowpass filter
3
BEGIN LOOP A - For each steering direction do:
4
BEGIN LOOP B - For each search angle do:
5
BEGIN LOOP C - For each node m = 1:M do:
6
Create input seen by node m
7
Lowpass filter data
8
Delay signal according to location in array
9
END LOOP C - Last node?
10
Yes
Decimate and apply window to delayed signal data from each node
11
Sum data & calculate beam power for search angle
12
END LOOP B - Final search angle?
13
Yes
Power for search angle
14
Final steering direction?
15
Yes
End program
NO
No
NO
Key:
Process
Loop limit (start or end)
Over head
Output

Figure 5.4.1 : Delay-and-Sum with Interpolation Sequential Algorithm
This flow chart shows the execution path for the delay and sum with interpolation sequential beamforming algorithm.

Next, the coefficients for the lowpass filter required for interpolation are calculated. The lowpass filter used in this design is a FIR (finite impulse response) filter, which is described in the Beamforming Theory background section.

The next step is to choose a particular steering direction by calculating the appropriate delay for each node. To measure the beam pattern for this steering direction, a source is placed at a number of different angles and the output power of the array is calculated for each input source. For example, assume that after input interpolation, α steering directions are available. The configuration file indicates how many different "search angles," named β, to calculate for each steering direction. With more search angles, a more resolute beam pattern can be calculated. For each of the α steering directions, the calculated power from the array is measured for β different search angles. This is equivalent to placing a single source at each of the β search angles and measuring the power from the array steered to a certain direction. Figure 5.4.2 shows a typical output from the sequential algorithm.

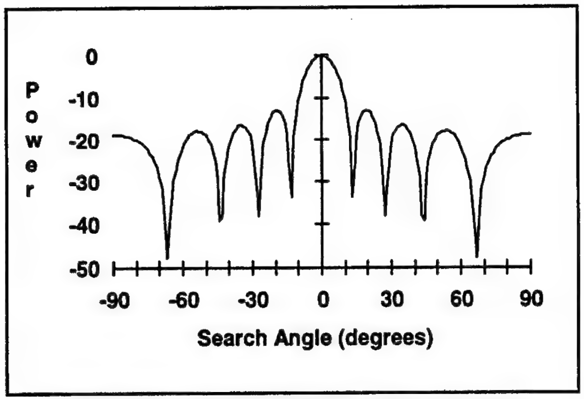


Figure 5.4.2 : Expected Beam Pattern Output

This is the expected beam pattern output of a simple linear array steered to 0 degrees.

Loops are used to try each of the different steering directions and search angles. In the block diagram, the first loop is the number of steering directions, the second loop is for each incoming signal, and the third nested loop is to calculate the data at each node for the incoming signal when seen from a particular steering direction. At this level, the data for the signal is found and then interpolated with the lowpass filter function.

After filtering the data, the signal at each node is delayed an amount relative to its position on the array. Once all the nodes have calculated their data for that source, a window function is applied and the data is summed and squared to form the beam power for that particular input signal. The process for a single beam pattern continues until all specified incoming angles are exhausted. Then the pattern begins again for a new steering direction and a new beam pattern.

5.4.2 Parallel Algorithm

To explore the parallelism in the DSI algorithm, the decomposition methodology summarized in Table 2.4.1 in the Parallel Decomposition background section is used as a guideline. The first step is partitioning the algorithm, which involves breaking the algorithm into small pieces, regardless of communication and architecture.

Perhaps the most intuitive partitioning of the DSI algorithm from the block diagram is functional decomposition. In Figure 5.4.1, blocks 1, 2, 3, 7, 8, 9, 11, 12, and 14 are considered the tasks. A functional breakdown of this algorithm imitates a pipeline structure, where each process is responsible for a single task. The data is streamed through the tasks until the algorithm is complete. Separating this algorithm into 9 tasks is a fine- to medium-grained functional approach. A fine-grained approach would be at the instruction level.

In domain decomposition, the data is divided into pieces, and individual tasks perform the same computations on these pieces of data. Because each node collects data, this algorithm is naturally domain decomposition. The data is divided among the nodes and each set of data is processed in the same way. At the end, the data is collected into a single entity and processed, and the result is given.

One partitioning that particularly stands out for this algorithm is a combination of the functional and domain. Figure 5.4.3 illustrates this concept. While the unrolling of the loops presents a domain decomposition, the interior of the loops where each task is performed by a different process (7, 8, and 9) represents a functional decomposition.

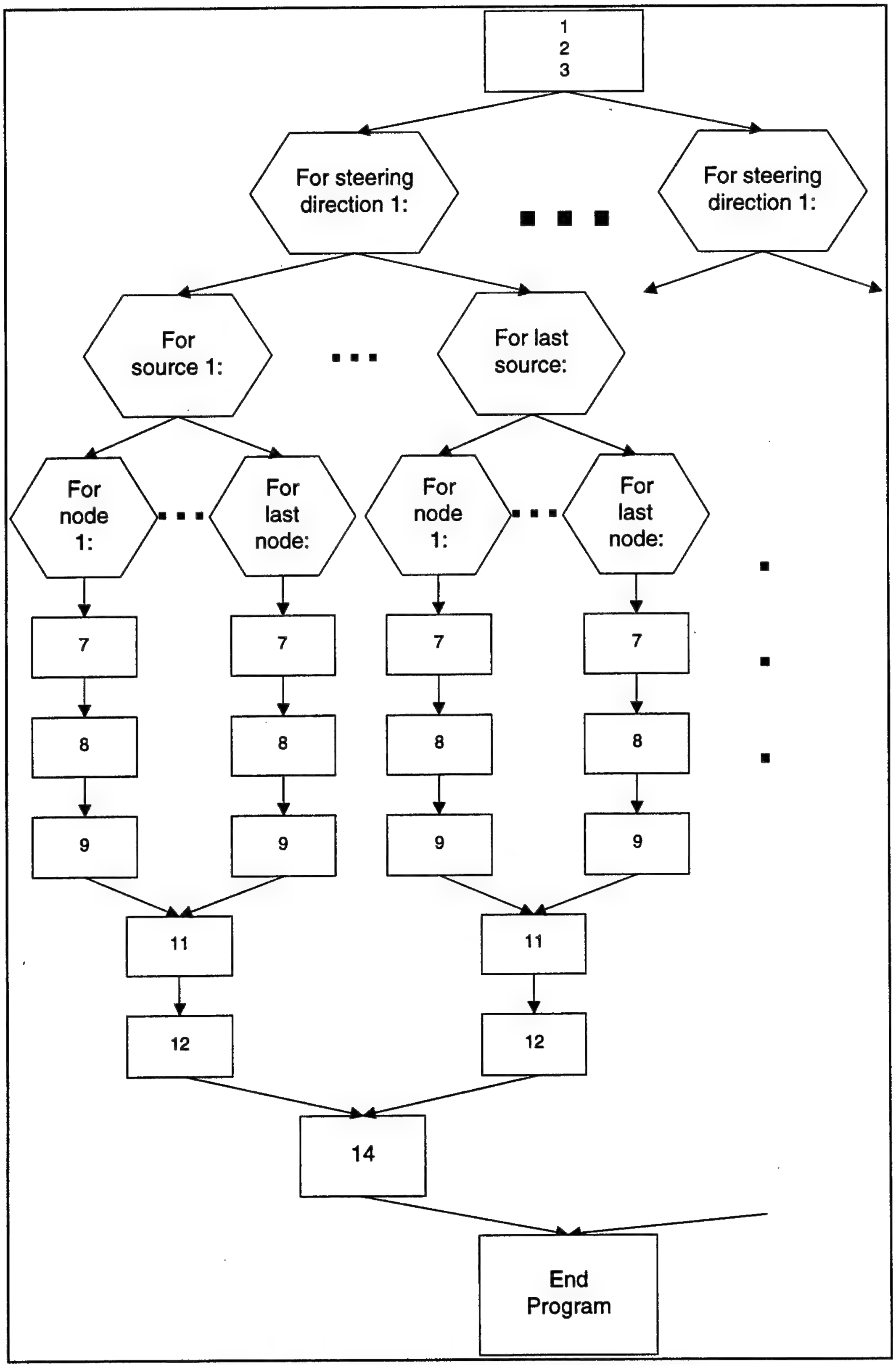


Figure 5.4.3 : Combination of Domain and Functional

Unrolling the loops represents domain decomposition, the interior of the loops represents functional decomposition.

The second phase in the program development is to investigate the communication required for the partitioned algorithm. In Figure 5.4.3, tasks 1, 2, and 3 were already grouped because the communication to open the file would be too extensive compared with the actions taking place in the tasks. Another possible bottleneck in the network might occur between tasks 7, 8, and 9 because the calculations are almost at the instruction level. Finally, a bottleneck could occur towards the end of the program when each partition is trying to consolidate its data to calculate the beam pattern for a particular steering direction. As many as 128 nodes could be trying to send data for each source direction. If each direction from -90 to 90 degrees were being investigated, this would cause 128x181 nodes trying to write their beam powers to the pattern at very close time intervals.

To address these communication concerns, the partition is agglomerated into a feasible design based on the architecture of the target system. The array is a message-passing multicomputer with either a unidirectional, bidirectional, or ring topology. In the partitioned algorithm, the collection of data at the end all flows into a single process.

For the unidirectional and ring topologies, this is a problem. The data can only flow in a single direction, thus restricting which process can perform this task. In addition, care must be taken when passing data to the next process so that its input queue does not become overloaded with data. One way to reduce the chance for queue overflow is to perform the loop for each steering direction sequentially instead of in parallel. This subtracts one level of parallelism, but reduces the buildup of data along the array. Another way to reduce data flow is to perform the summing as the data is moved along the array instead of at the very end of the algorithm. With these two data flow reductions, the algorithm should be ready to apply to each of the topologies. Finally, tasks 7, 8, and 9 are grouped into a single process, which makes the algorithm a domain decomposition algorithm.

The unidirectional array is the simplest to implement of the three because the virtual architecture of the array remains static. Only one end process is ever responsible for the final result, and that process is labeled the "boss." If a break occurs in the array, the new end process takes over, thus giving the array fault tolerance. Figure 5.4.4 represents the unidirectional algorithm. The first time through, each process is waiting for data from the previous node before it begins any calculations. In this way, the algorithm resembles a pipeline. After each process has received some data, all processes are busy until the next steering direction is encountered, at which time they synchronize. This is necessary to keep a build up in the data at nodes further down the line.

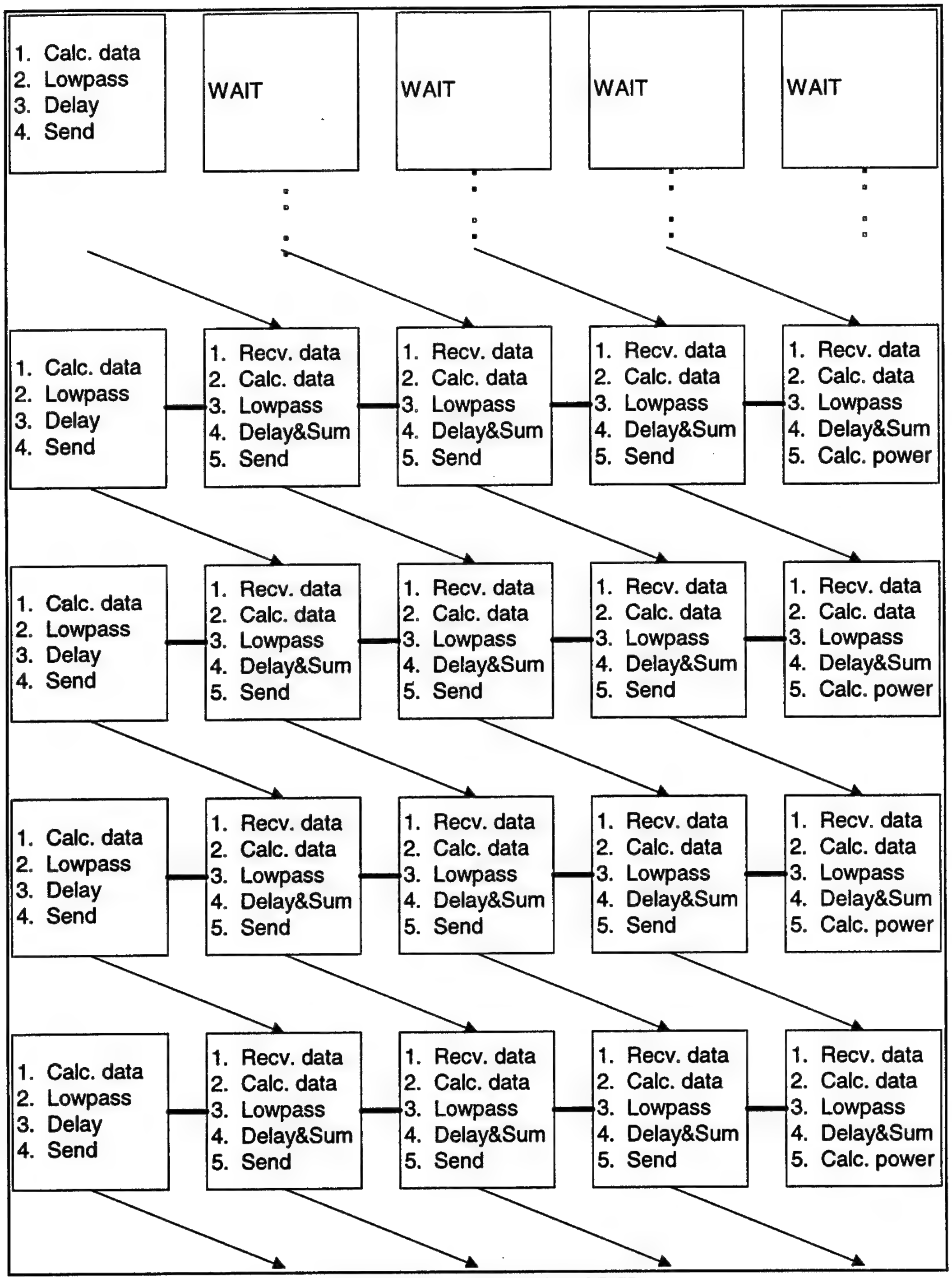


Figure 5.4.4 : Unidirectional DSI

As can be seen by the uniform direction of the arrows, this flow chart depicts the unidirectional delay and sum with interpolation.

The ring topology is similar to the unidirectional array in that it is also unidirectional. In fact, the ring can be viewed as a modification to the unidirectional array. Unlike the previous algorithm, the final process (boss) that calculates the power does not remain the same node. The responsibility of collecting the final data floats around the ring to different processes. If the current boss is process 1, where the ring is configured such that the numbers increase clockwise and the direction is clockwise, then the next boss is process 0. This way, while the boss is calculating the beam power for one angle, the previous node can set up as the boss and begin to receive data.

The bidirectional array is somewhat different from the ring and unidirectional. Since the bidirectional array can communicate in both directions, we can assign the middle node to be the boss. If there is an even number of nodes, then the node closest to the transmitter becomes the boss (closest to node 0). All nodes to the left of the boss send data to the right, and all nodes to the right of the boss send data left. This algorithm takes less time to process the data because data is being sent in two directions at once and only half as far. Thus, more results are available quicker. Figure 5.4.5 illustrates the bidirectional array.

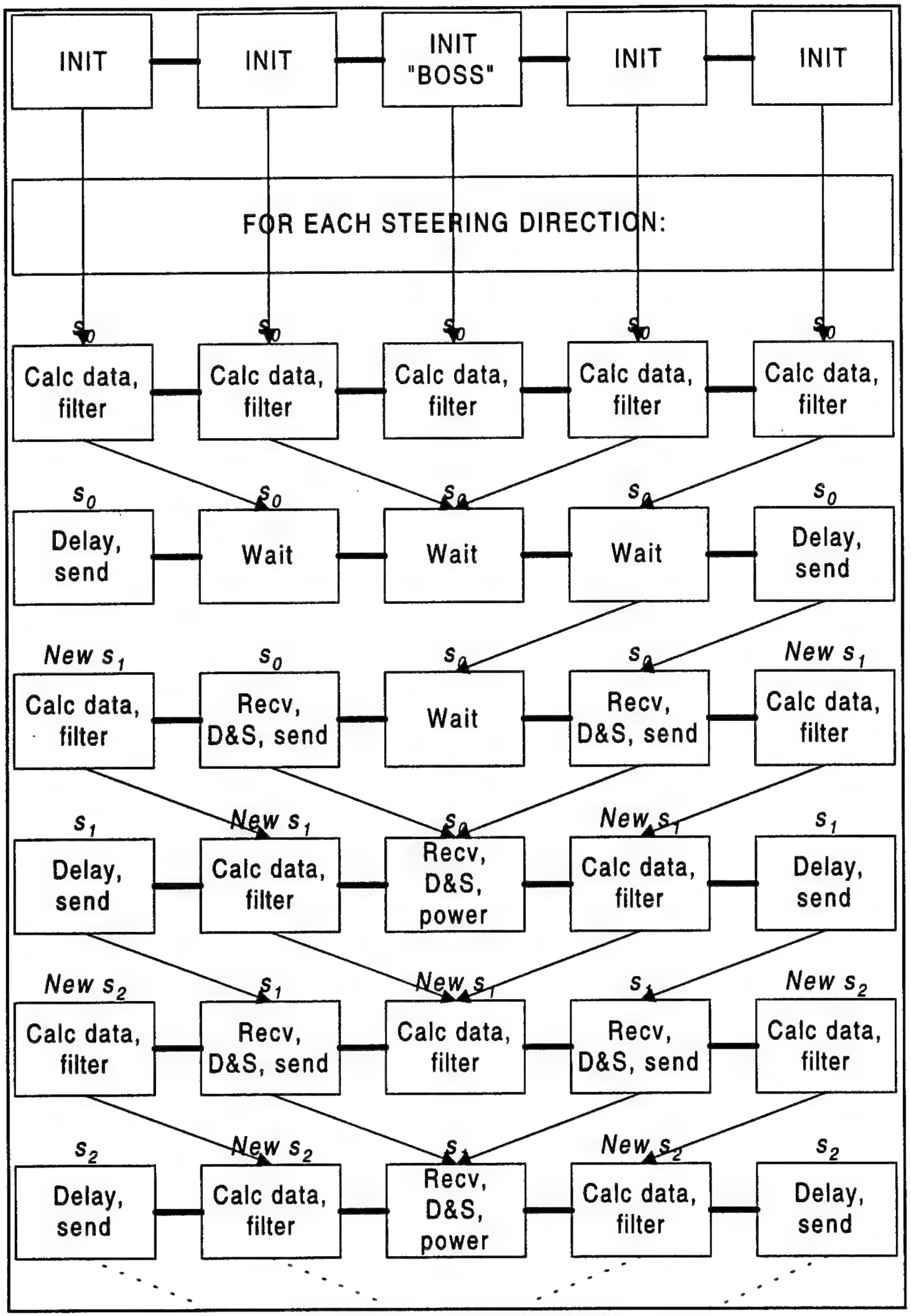


Figure 5.4.5 : Bidirectional DSI

The s_n variables represent the incoming angles. Downward motion represents increasing time. Each column represents one node. The boss is responsible for calculating the power and beam pattern.

The final phase in program design is the mapping. In reality, each node will have its own processor such that the algorithms map directly to the architecture. Theoretically, the bidirectional array should provide the best performance since data only has to be sent half as far as the other nodes. The disadvantage to the bidirectional array is the extra cost involved in setting up such a configuration. The next best algorithm is the ring, then the unidirectional. The extra overhead incurred in the ring should not have much of an effect on the performance because it does not require much to recalculate who the boss is. With the trade-off between overhead from the ring and the calculation time in the ring, the ring and unidirectional topologies are probably close in performance. In order to test this theory, the three topologies are simulated and the results provided in Chapter 7 (Preliminary Software System).

5.5 Frequency-Domain Decomposition

The second major sub-category of algorithms researched was basic frequency-domain beamforming. Specifically, the algorithm used was a standard radix-2 Fast Fourier Transform beamformer. Rather than concentrating on decomposing the algorithm in several ways or decomposing a single iteration, the FFT beamforming algorithm was used as the representative algorithm for several simulation methods; therefore, only a limited number of decomposition techniques are used. Furthermore, the frequency-domain algorithm works strictly as a detection algorithm in that it searches for a fixed number of sources in the search angle range.

5.5.1 Sequential Algorithm

The first step in any algorithm parallelization project is to examine the sequential algorithm. The first computation in the FFT beamformer is to calculate the windowing functions for the various nodes and to multiply that node's data samples by that factor. Next, the algorithm takes the Fast Fourier Transform of the windowed data, node by node. The algorithm then enters a loop that executes once for each steering direction. For each iteration of the loop, the algorithm multiplies the transformed data from each node by the node-dependent value for the steering factor, which is based on the node's location. The next step is to sum all the data, sample by sample, for all 32 or 64 samples. The algorithm then inverse transforms this summed data. Since the result is a complex vector, a real, single-valued result is obtained by computing the magnitude of each point and averaging the points. This procedure results in a single value that is the signal power seen by the beamformer from the current steering direction. The algorithm stores this value before looping to the next steering direction. At the end of the algorithm, all the values obtained from the steering directions can be plotted versus steering direction so that the signal power can be seen spatially by the user. Furthermore, the FFT algorithm allows an additional independent variable (incoming frequency) and the associated results to be observed in a manner similar to Figure 5.5.1. The entire process is repeated for successive iterations with new sensor input values. Figure 5.5.2 shows the flowchart for the sequential algorithm.

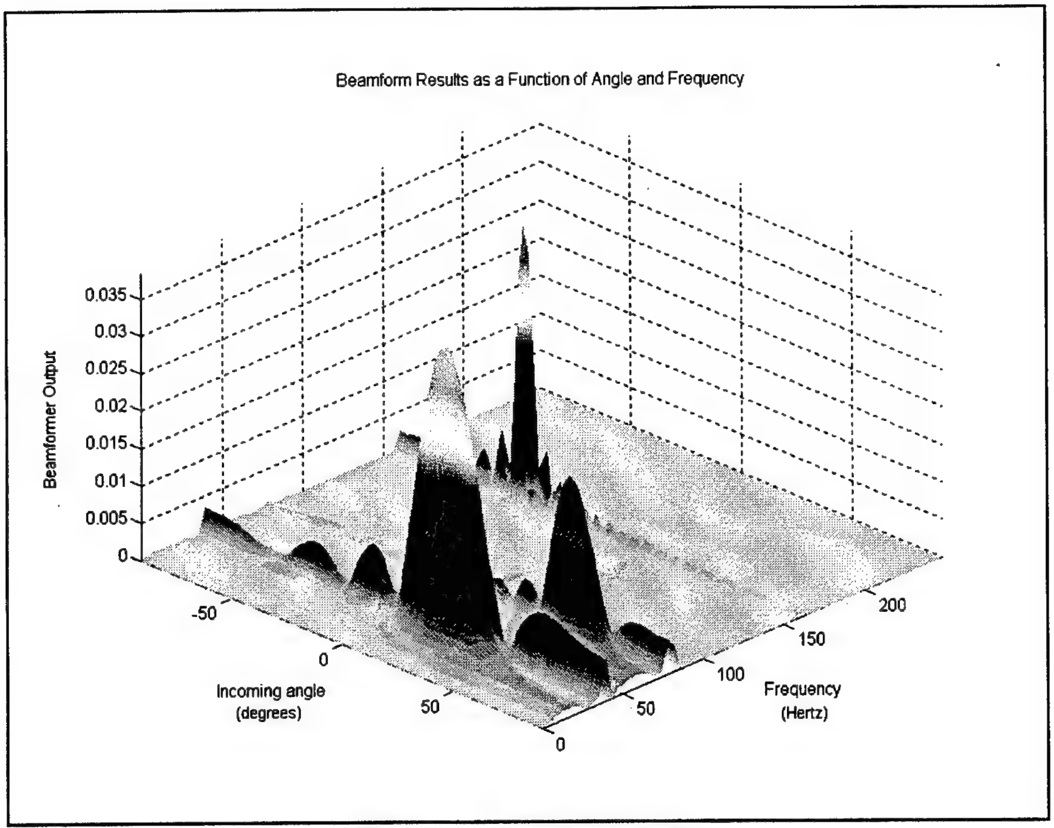


Figure 5.5.1 : Beamformer Output

This screen shot from the MATLAB GUI shows the capabilities of the FFT beamformer. Along the two horizontal axes is the signal frequency and signal incoming direction.

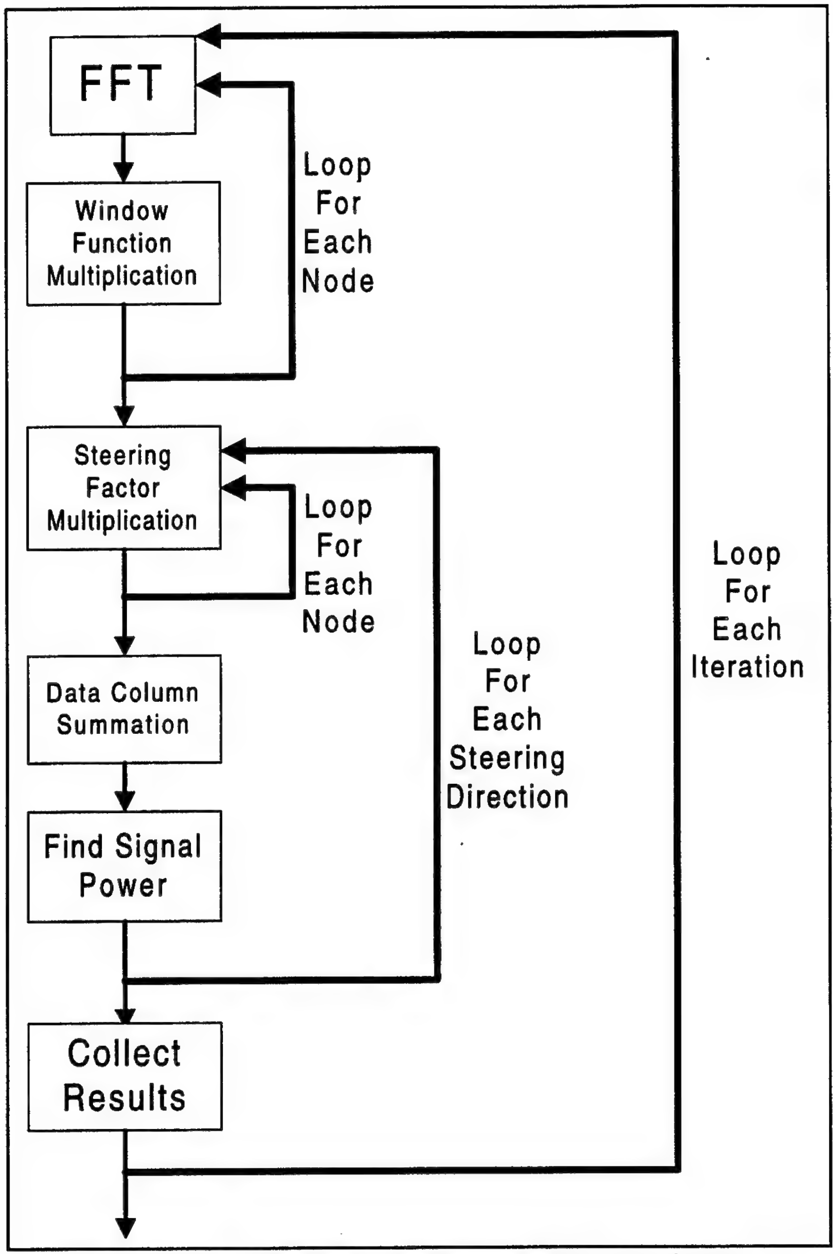


Figure 5.5.2 : Flowchart for the Sequential FFT Beamformer

The flow chart above describes the purely sequential FFT beamformer. The processor performs loops for each node and loops for each steering direction, all within a loop for each iteration.

5.5.2 Parallel Unidirectional Linear Array Algorithms

The first parallel algorithm created in the frequency-domain realm of beamforming was targeted to run on a unidirectional linear array and is called the parallel unidirectional FFT, or PUF. Due to the unidirectional nature of the underlying network in this algorithm, there is a limited amount of parallelism possible. Since there must always be a summation of all of the nodes' data, there must be a single node computing the sum, and that node must be the final node downstream. Due to this fact, the array is not evenly balanced as far as computational load is concerned. The most downstream node must always do more work (which includes the summations) than any of the other nodes, thus limiting the possible parallelism. Amdahl's Law of Parallel Computing states that the maximum possible speedup of an algorithm is limited by the amount to which that algorithm is inherently sequential in nature [HWANG93]. With PUF's limitations on parallelism, the speedup of the parallel unidirectional FFT beamforming algorithm is not expected to be linear with the number of processors.

The first version of the parallel unidirectional FFT decomposition is *PUFv1*. The paradigm used for this first version is data parallelism, which is simply stated as parallelism obtained by doing the same operation on different data at the same time. Since the FFT of one node's data does not depend on the transform results of any of the other nodes, all transforms can be done simultaneously in a data-parallel manner. This is the first case where processing array nodes are needed, because each node can do its own FFT without communication. Additionally, the node windowing factor is a function only of the chosen window and the node's position, not of the results of any other computation on some other node; therefore, the computation of the windowing factor and its multiplication with the data values for that node can be done in parallel along with the transforms. Once each node has completed its local computations, the nodes send their data down the linear array to the first node. The first node then carries out the remainder of the sequential algorithm, which means it executes the loop for all the different steering directions, which includes the multiplications for the steering factor and the inverse FFT's. Finally, the front-end collects all the results from the steering directions. Figure 5.5.3 shows the flowchart for how the parallelism in this algorithm takes place, where blocks lined up horizontally are computed simultaneously.

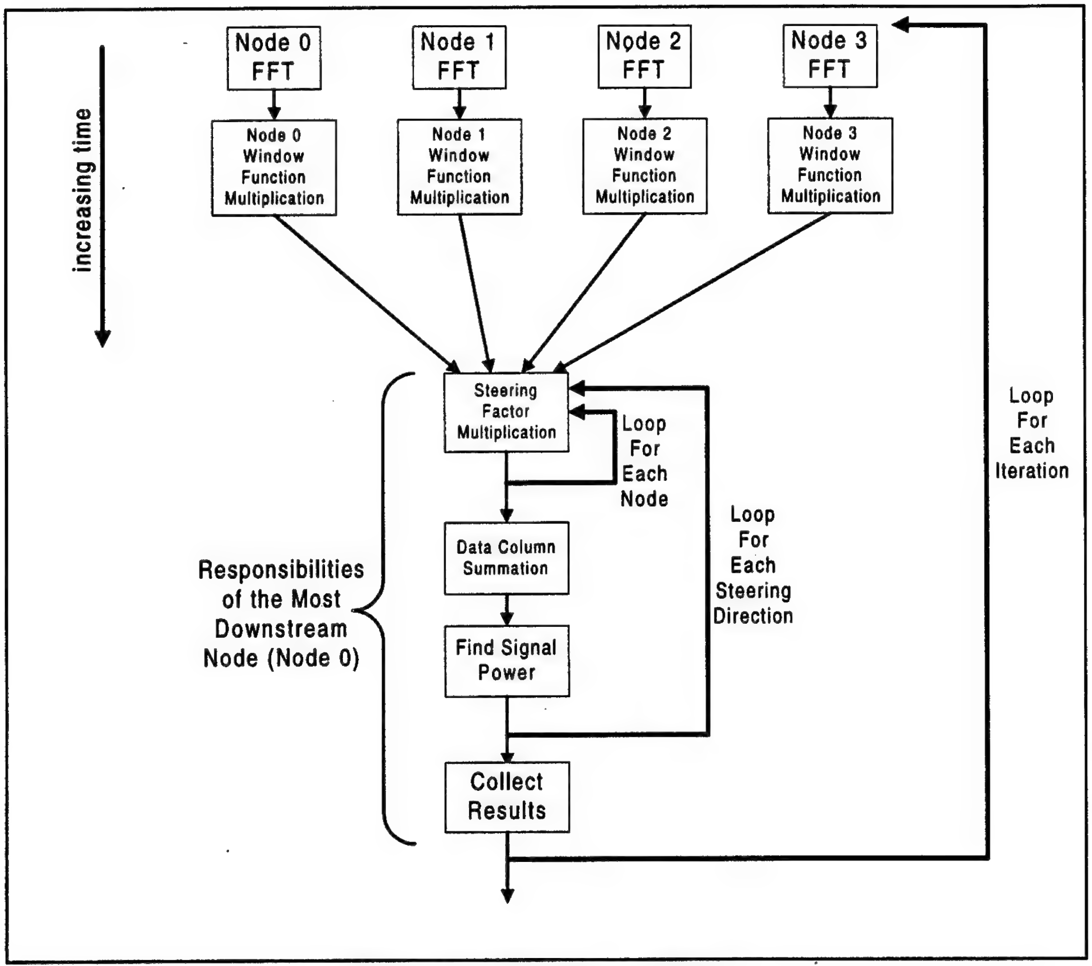

Node 0 FFT
Node 1 FFT
Node 2 FFT
Node 3 FFT
increasing time
Node 0 Window Function Multiplication
Node 1 Window Function Multiplication
Node 2 Window Function Multiplication
Node 3 Window Function Multiplication
Steering Factor Multiplication
Loop For Each Node
Loop For Each Steering Direction
Loop For Each Iteration
Data Column Summation
Find Signal Power
Collect Results
Responsibilities of the Most Downstream Node (Node 0)

Figure 5.5.3 - Flowchart for *PUFv1*

This flow chart depicts how the parallelism in PUFv1 is inserted in the FFT and windowing factor computations.

The next decomposition of the parallel unidirectional FFT, *PUFv2*, is meant to improve on the small amount of possible parallelism of the previous version. One important aspect of the previous algorithm is the communication pattern; the amount of communication grows linearly with the number of array nodes. To elaborate, as more array nodes are added, the increase in the amount of communication done is equal to the percent increase in number of array nodes. This is due to the fact that the front node can receive from only one node at a time, and all receives must be made one after another. The nodes cannot combine their data ahead of time for the front node because the first node must multiply each column by node-dependent factors. The *PUFv2* algorithm moves the steering direction loop from the first node and makes the operations in the loop part of the responsibilities of the several nodes. The major goal this accomplishes is to complete the summation as the communication is progressing. Rather than sending column after column to the front node, each node receives a column from upstream, adds its column to it, and sends the single summed column downstream. There is considerably less communication with this approach, which is illustrated in Figure 5.5.4. There is, however, a drawback in that the new form of communication (with summation) must be carried out for each steering direction. There is a trade-off between an algorithm that communicates a large amount of data but only doing it once each iteration, and communicating considerably less values but doing it 91 times per iteration for 91 steering directions.

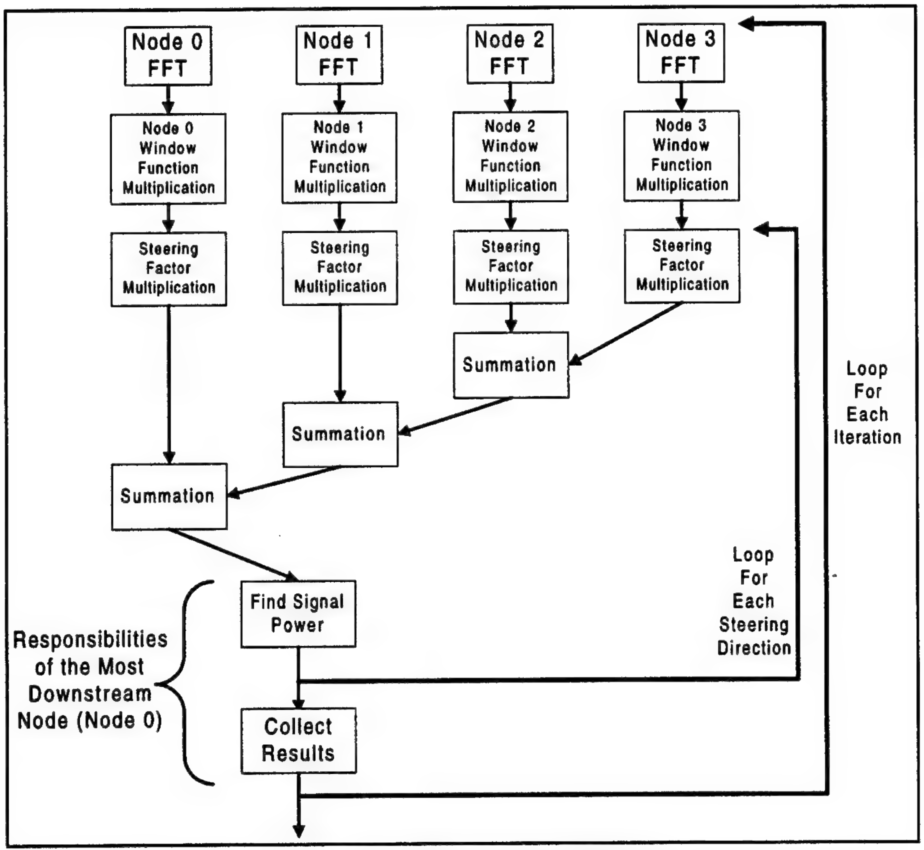


Figure 5.5.4 - Flowchart for *PUFv2*

The PUFv2 flow chart demonstrates the lighter communication because each node sums data from the previous node so that it only has to send one column.

5.5.3 Parallel BDN Algorithms

The next algorithms were developed with underlying architectures other than a unidirectional linear array, which allows for more efficient programs. The algorithms were developed for a fully-connected bounded-degree network (BDN), which means that any node can communicate with any other node by way of some communication path through the nodes. The algorithm no longer needs to be concerned about which nodes can communicate with which other nodes. The algorithms developed which allow for this are the parallel ring FFT (PRF) and the parallel bidirectional linear array FFT (PBF). In a ring configuration, a node can communicate with any other node by sending down the ring, and the data will hit the destination at some point before a full cycle of the ring is completed. In a bidirectional linear array configuration, a node can send either up or down the linear array so that any node can be the recipient. At this point, the exact method is not important for the algorithm. What is important is that full communication is possible. As opposed to the unidirectional case where the first node always had to do more work than the other nodes, the algorithms for the fully connected network allow the node doing the front-end work (the boss) to be floating. Rather than forcing all data to be sent to the first node every time, the data can be sent to a different boss node each iteration. The concept is that of a floating front-end processor, where each node uses a deterministic algorithm to figure out which array node is acting as the front-end for a particular iteration. This ability makes possible better link utilization and better processor computational usage. The back-end node no longer has the least work to do because on the next iteration, it may be designated the front-end to which all other nodes send their data to be manipulated.

In addition to data-parallelism, these algorithms introduce a type of agenda parallelism, which shall be called here *specified agenda parallelism.* In agenda parallel applications, the working processes figuratively reach into a grab-bag of tasks which need completion. When they pull out their assignments, they get to work on them. In the *specified* agenda parallel concept, the tasks which the working processes pull out of the grab-bag are not random choices. Instead, a selection algorithm is created so that the tasks pulled out of the bag are in a specified order.

These algorithms take the algorithm from PUFv1 and implement the floating front-end. The nodes all take their own transforms and compute their own windowing function factor to multiply into their data. Then each node determines who is the front-end by using a specified formula. This formula simply follows a "round-robin" scheme in which the front-end for the next iteration is the neighboring node to the front-end for the current iteration. For additional iterations, the front-end which is chosen follows a path around the available nodes. Once a front-end is chosen for a particular iteration, communication then proceeds down the ring or through the bidirectional linear array to that front-end just as it did in PUFv1. The front-end node then starts the loop for the several steering directions and puts together the results.

The algorithm follows the tradition of pipelining, where one operation does not need to be completed before the next operation is started. Pipelining applies to this algorithm in that the nodes can collect another group of signal samples from the audio transceivers and begin the second iteration before the first iteration is completed. At the beginning of the second iteration, the node doing the front-end work stops what it is doing for just enough time to FFT the second iteration data, multiply by the windowing factor, and send it off to the second-iteration front-end. Once the data has been sent to the new front-end, the node resumes front-end work for the previous iteration data. The process continues for the third iteration, where the two front-ends from the previous two iterations stop their work for a moment. As the iterations progress, more and more nodes are doing front-end work for their respective iteration numbers, all in different stages of completion.

It is not desired for the algorithm which determines the next boss to assign a node which is still working on front-end work from a previous assignment; therefore, a mechanism must be set up to assure there is no wrapping of the front-end assignment until such time as a complete iteration of work can be completed. This is done by having the front-end processors break to start the next iteration after a specified number of steering direction loops. For example, if there are 32 nodes scheduled to be front-end nodes and

91 steering directions to check for each iteration, then the front-end nodes will stop their front-end work after every three executions of the steering direction loop. Therefore, by the time the assignment of front-end nodes has wrapped around to a node already assigned, that node will have finished the 91 steering directions (since 3x32 equals 96) and be free to take on responsibilities of another assignment.

There is some inefficiency in this algorithm since the regular intervals which the front-ends use to take time to start the next iteration introduces some extra delay. In the above example, where it takes the equivalent of 96 iterations of a steering direction loop to reassign a node as a front-end, the nodes only execute 91 steering directions, so they are waiting to be reassigned for the equivalent time of 5 steering direction iterations. This inefficiency can be avoided by creating an algorithm that uses two threads for each node. The first thread is in charge of communications for that node, and the second thread carries out any front-end work assigned to it. The communication thread waits until exactly the appropriate time to send data for another iteration so that the iterations occur in just enough time so that a front-end thread will finish just before being reassigned. There is a significant problem with this method, however, because the additional threads introduce a heavy overhead cost. The additional time taken to organize the additional thread for each node more than overshadows the time gained by saving the time for those 5 steering direction iterations; therefore, the first version without the additional threads is used to represent the best ring algorithm. The second version will be useful in porting the algorithm to the BONeS simulation. This is because the BONeS simulation is not done in real-time, so the additional thread overhead will contribute nothing to the time steps of which the simulation keeps track.

* * *

The development of the generalized message-passing system model allowed the rapid algorithmic prototyping of beamforming algorithms with the understanding that the message-passing algorithms developed would be portable. Once in the abstract model, the beamformers could undergo standard parallel decomposition techniques. Likewise, simulation and implementation of the algorithms is reasonably straight-forward since the simulator is designed to offer an opaque message-passing system and the actual sonar array will have similar message-passing hardware.

6. Simulator Development

Researchers in the team have established a sequence of steps for the development of baseline and parallel programs and for the presentation of results from those programs. These steps include implementation of an algorithm for baselining and result verification in MATLAB. Next, the researcher converts the MATLAB program to C or C++ code and parallelizes the code according to an appropriate decomposition method. The C or C++ programs are then run and timed over one of the laboratory's testbeds. Ultimately in the second phase of this project, the parallel program is linked to the Block Oriented Network Simulator (BONeS) for detailed simulation over a high-fidelity-simulated sonar array architecture. A number of resources were created to make progression along this development track simpler. These resources include standard configuration files for program and architecture parameters, an object-oriented algorithm framework for easy implementation of parallel algorithms into written code, and a number of graphical user interfaces for handling parameters and indicating results. This development process is illustrated is Figure 6.0.1.

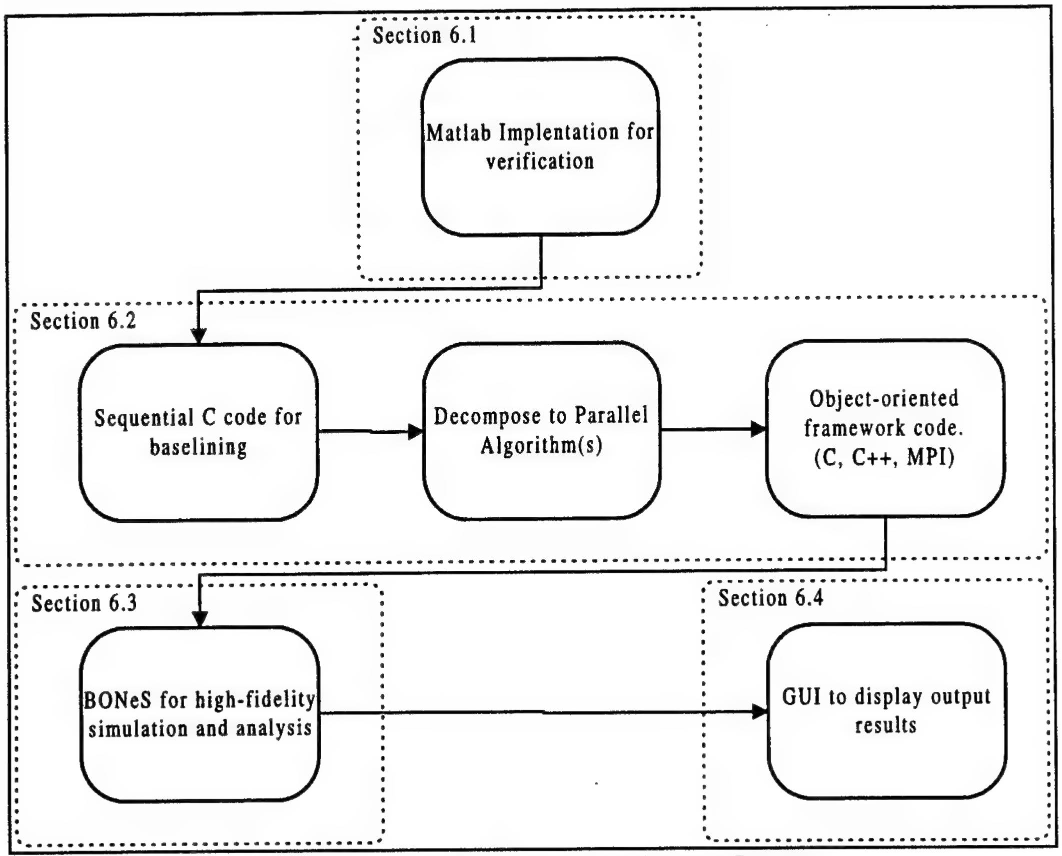
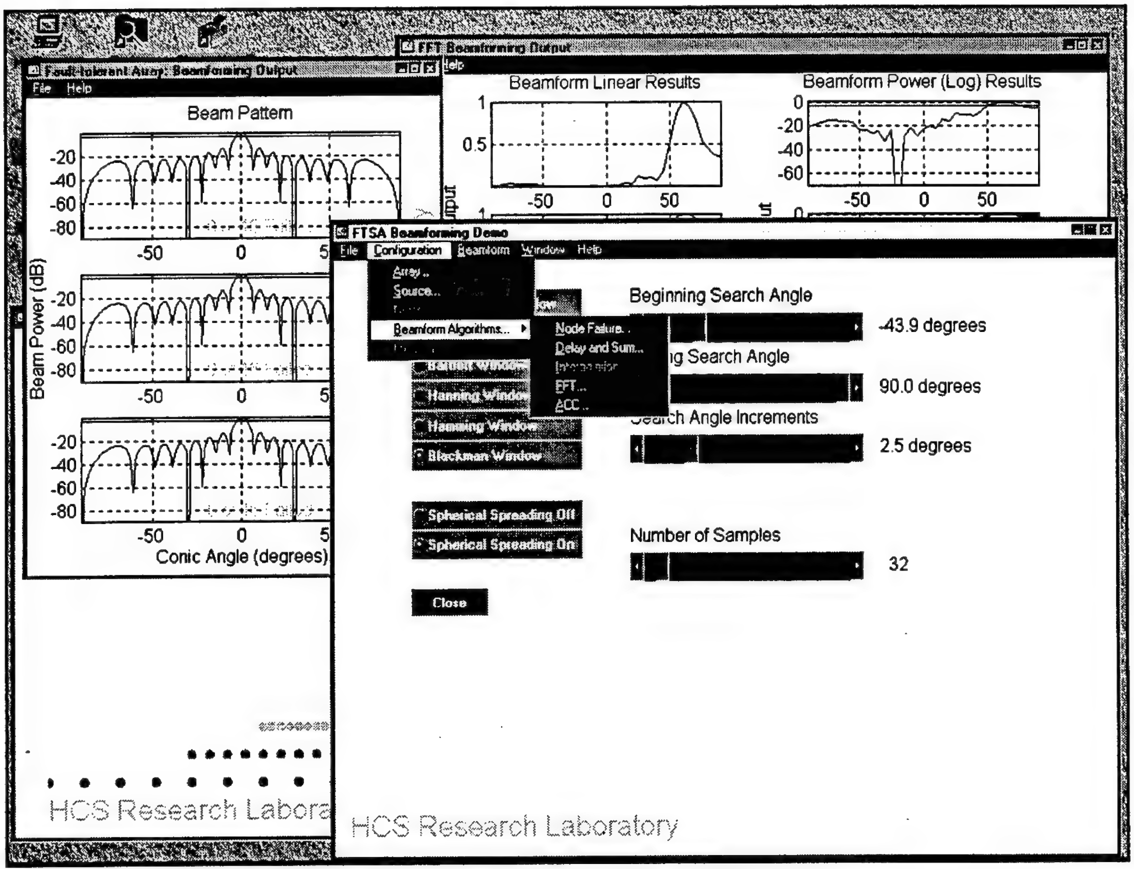

Section 6.1
Matlab Implentation for verification
Section 6.2
Sequential C code for baselining
Decompose to Parallel Algorithm(s)
Object-oriented framework code. (C, C++, MPI)
Section 6.3
BONeS for high-fidelity simulation and analysis
Section 6.4
GUI to display output results

Figure 6.0.1 : Parallel Program Development Process

This figure depicts the four major steps (in dotted boxes) which have been and will be followed for developing parallel code for the sonar array. The BONeS simulator and the final GUI are slated for development in the near future.

6.1 MATLAB Implementation

The first step in the program development track is implementation in MATLAB, for which a graphical user interface has been written, shown in Figure 6.1.1. The GUI provides a number of conveniences so that the task of coding and verifying the beamforming algorithm is made easier. First, the MATLAB GUI provides all array and source parameters, such as node spacing, sampling frequency, and source frequency, in global variables so that any program can effortlessly access them. Furthermore, the GUI provides standard windows-style check boxes and slide bars for simple user input of all parameters. To prepare an algorithm for use in the MATLAB environment, the programmer is concerned only with coding the algorithm; the interface is already available. The MATLAB GUI also supports three overlapping nested arrays, each with its own controls for spacing and number of nodes. A window can be opened to show a graphical depiction of the layout of the nodes of the three nested arrays, as shown in the bottom left corner of Figure 6.1.1. Next, the GUI can run an algorithm chosen by the user and display the beamforming results, which can be preserved in a separate window while additional algorithms are run. Another feature of the GUI is a fault-injection option, which allows the user to choose how many nodes in the array will fail, and the beamforming output from one of the algorithms will display the results of the failures. The MATLAB GUI will also save the parameters set by the user to a file for retrieval at a later time, making it simpler to compare the results of several algorithms over a number of sessions. A flow chart showing how the GUI interface connects several of the entities is given in Figure 6.1.2.



Beginning Search Angle -43.9 degrees

Ending Search Angle 90.0 degrees

Search Angle Increments 2.5 degrees

Number of Samples 32

Figure 6.1.1 : Screen Shot from the MATLAB GUI

This screen shot shows several of the functions of the MATLAB GUI, including node failure (top-left), FFT results (top-right), graphical node spacing (bottom-left), and FFT setup screen (bottom-right).

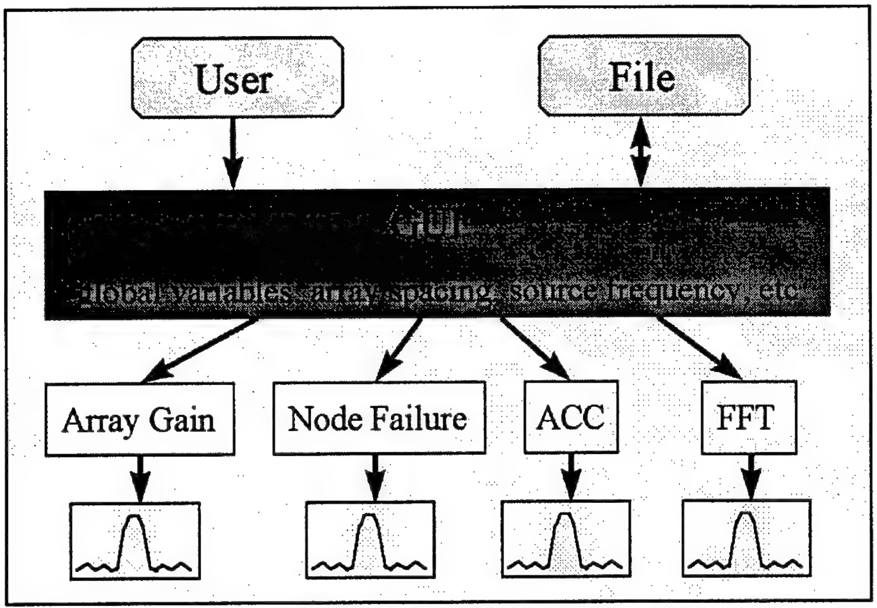


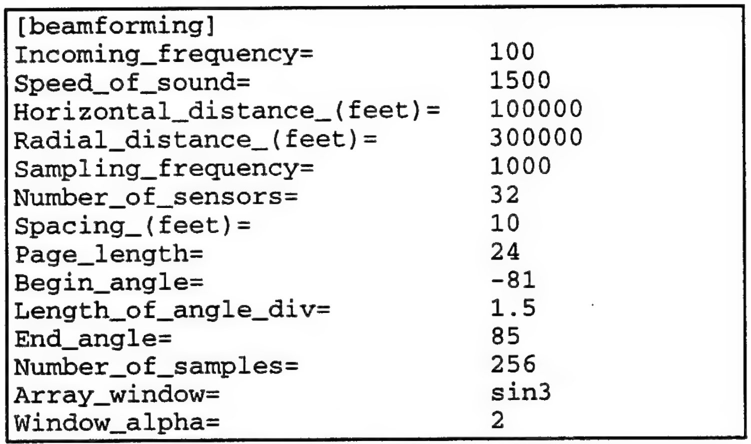
Figure 6.1.2 : MATLAB GUI Interface Flow Chart

This figure depicts how the user can interface with configuration files and the several algorithms implemented in MATLAB.

6.2 C/C++/MPI Program Conversion

The next step in the development track for algorithms, a step for which a number of tools have been created to assist, is conversion to C or C++. This step also involves creation of an MPI parallel program and the timing of the baseline and the parallel program over an available network interconnect testbed. Among the tools developed are a standard configuration file format, ready-made C functions for reading and writing to the configuration files, an object-oriented framework for programs, and a text-based runtime system.

The standard configuration file format was developed to easily support the parameters of any program. Parameters are stored in an editable ASCII file in the format "`parameter=value`" so that users can change any parameters with a standard text editor. Furthermore, users can save a configuration file whose parameters were of some particular interest into a new file. A representative configuration file is shown in Figure 6.2.1. Besides supporting parameters which appear once, such as sampling frequency and windowing function, the file format also supports multiple signal sources, so that there can be parameters for the incoming frequency and direction for a number of signals. Though different algorithms require different parameters, all parameters for all programs may be stored in the same configuration file because parameters not needed in a particular program are simply skipped when the program reads from the file. Furthermore, when a program writes all new parameter values to a file, only those parameters which the program has been using are overwritten, leaving unchanged the values which other programs may use. In order to facilitate implementation of the standard configuration file format into all the parallel and sequential programs being developed, a number of library functions were written. These functions provide a program with the ability to read configuration files, write configuration files, and change individual values.



```
[beamforming]
Incoming_frequency=              100
Speed_of_sound=                  1500
Horizontal_distance_(feet)=      100000
Radial_distance_(feet)=          300000
Sampling_frequency=              1000
Number_of_sensors=               32
Spacing_(feet)=                  10
Page_length=                     24
Begin_angle=                     -81
Length_of_angle_div=             1.5
End_angle=                       85
Number_of_samples=               256
Array_window=                    sin3
Window_alpha=                    2
```

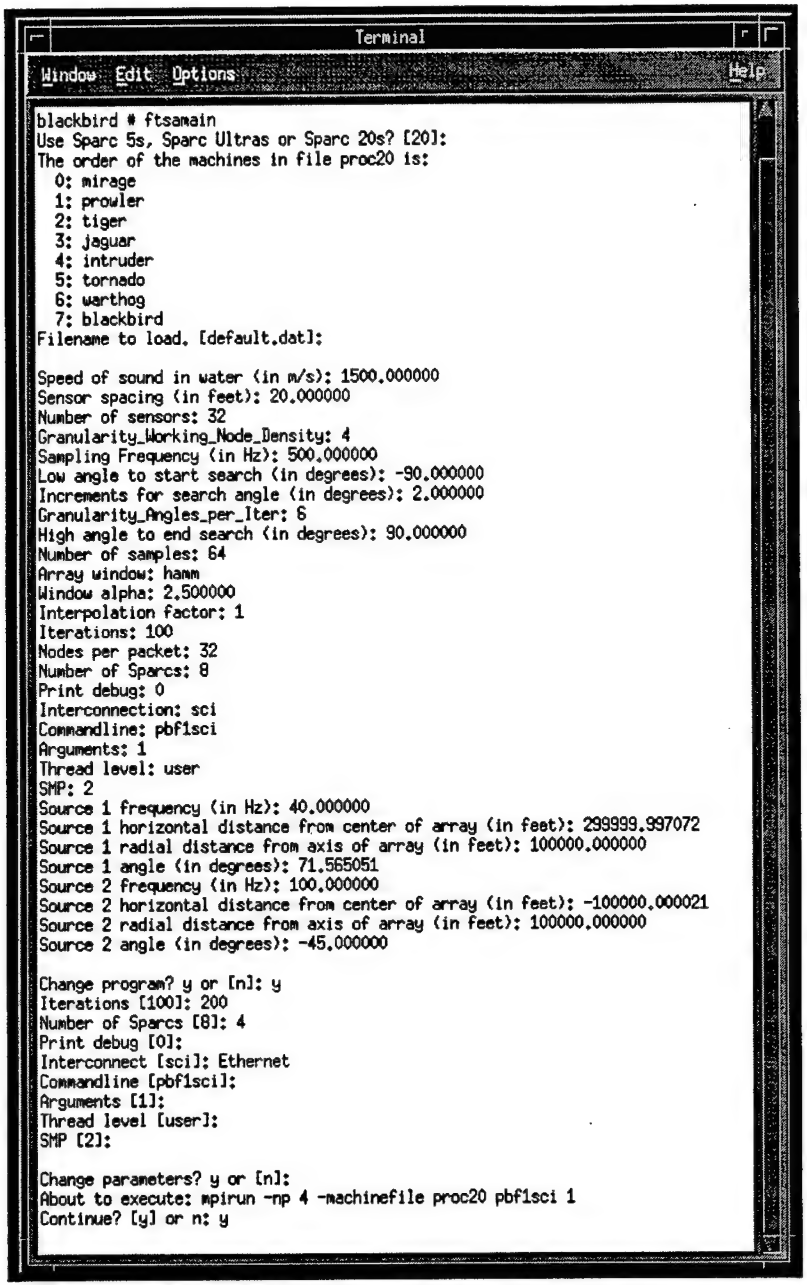
Figure 6.2.1 : Example Configuration File

This box presents the printout of a sample configuration file.

The next tool developed for help in the conversion of an algorithm to a coded parallel program is an object-oriented framework in C++. The framework is meant to provide a communication abstraction and to make transparent many of the book-keeping functions needed to code a sonar array. The framework provides object types for a sonar array, and a signal source. The sonar array object is composed of a number of node objects. Each node object is composed of a network interface object which manages the sending and receiving of data, a analog-to-digital converter object that provides samples of the input source, and a processing object that contains the specific algorithm implementation code. The interface to the array object allows parameters such as the number of nodes in the array, the node spacing, the sampling frequency, and the network topology to be set. Upon creation of an array object, the underlying framework automatically initializes each array node object as a single thread, initializes the MPI runtime system, and spreads the threads over the numerous workstations.

The object-oriented framework also provides communication abstraction that separates the programmer from the difficulty of implementing a message-passing system in the multithreaded, multicomputer environment. In the user-defined processing function (where the algorithm is implemented), the programmer can use generic send and receive functions to communicate with other nodes in the array. Send and receive requests are passed down the object hierarchy to the node's network interface object. The network interface object uses a set of queues and logic to manage incoming and outgoing data. Data intended for a node within the same workstation is sent via a memory copy and data intended for a node on a different workstation is presently handled using MPI. The two key benefits of abstracting the communication details to generic send and receive calls is that programmer can concentrate solely on developing the algorithm code and the actual implementation of the node-to-node communication can be changed and optimized without effecting the algorithm code.

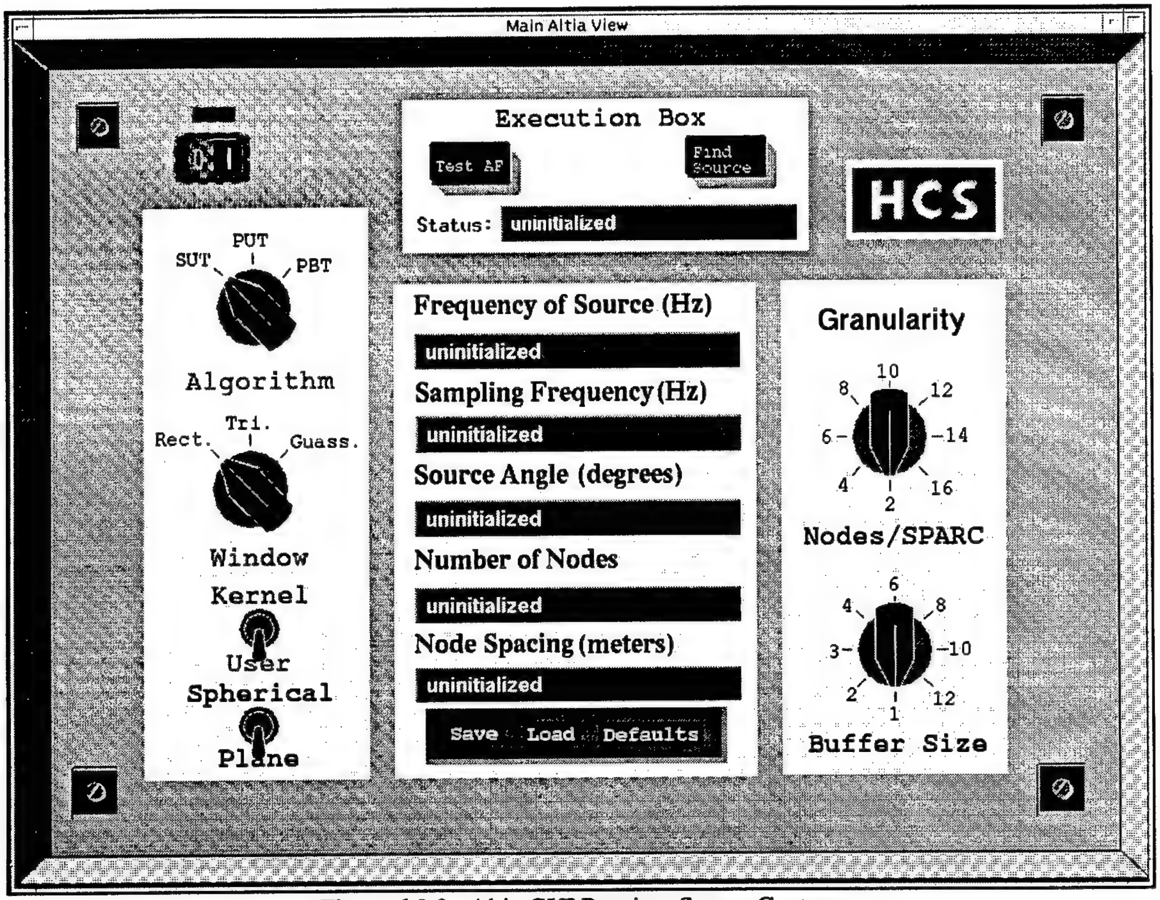
The last tool for use in the parallel coding step of the development process is a runtime system for the various programs. There is currently a text-based runtime system, and the graphical runtime system is currently in development. The function of the runtime system for the parallel programs and baseline programs mirrors many of the functions of the MATLAB GUI discussed above. The runtime system provides access to configuration files, allows parameters to be input, runs the program the user chooses, and reports program outputs such as the running time. The runtime system provides an interface for the user to the parallel processing happening on the testbed in the background. Figure 6.2.2 shows a screen shot from the text-based runtime, and Figure 6.2.3 shows a screen shot from the GUI runtime.



```
blackbird # ftsamain
Use Sparc 5s, Sparc Ultras or Sparc 20s? [20]:
The order of the machines in file proc20 is:
  0: mirage
  1: prowler
  2: tiger
  3: jaguar
  4: intruder
  5: tornado
  6: warthog
  7: blackbird
Filename to load. [default.dat]:

Speed of sound in water (in m/s): 1500.000000
Sensor spacing (in feet): 20.000000
Number of sensors: 32
Granularity_Working_Node_Density: 4
Sampling Frequency (in Hz): 500.000000
Low angle to start search (in degrees): -90.000000
Increments for search angle (in degrees): 2.000000
Granularity_Angles_per_Iter: 6
High angle to end search (in degrees): 90.000000
Number of samples: 64
Array window: hamm
Window alpha: 2.500000
Interpolation factor: 1
Iterations: 100
Nodes per packet: 32
Number of Sparcs: 8
Print debug: 0
Interconnection: sci
Commandline: pbf1sci
Arguments: 1
Thread level: user
SMP: 2
Source 1 frequency (in Hz): 40.000000
Source 1 horizontal distance from center of array (in feet): 299999.997072
Source 1 radial distance from axis of array (in feet): 100000.000000
Source 1 angle (in degrees): 71.565051
Source 2 frequency (in Hz): 100.000000
Source 2 horizontal distance from center of array (in feet): -100000.000021
Source 2 radial distance from axis of array (in feet): 100000.000000
Source 2 angle (in degrees): -45.000000

Change program? y or [n]: y
Iterations [100]: 200
Number of Sparcs [8]: 4
Print debug [0]:
Interconnect [sci]: Ethernet
Commandline [pbf1sci]:
Arguments [1]:
Thread level [user]:
SMP [2]:

Change parameters? y or [n]:
About to execute: mpirun -np 4 -machinefile proc20 pbf1sci 1
Continue? [y] or n: y
```

Figure 6.2.2 : Text-Based Runtime Screen Capture

This screen capture is from an execution of the text-based runtime system. It gives the user several choices. The user can hit the ENTER key to accept the default choice.



Status: uninitialized

Frequency of Source (Hz): uninitialized

Sampling Frequency (Hz): uninitialized

Source Angle (degrees): uninitialized

Number of Nodes: uninitialized

Node Spacing (meters): uninitialized

Figure 6.2.3 : Altia GUI Runtime Screen Capture
The Altia GUI provides many of the same functions as the text-based runtime system, only in a more pleasing style.

6.3 BONeS Simulator

The third step in the development track is to link the parallel programs developed into the BONeS simulation of the network architecture below. This is accomplished by an interface library currently under development which is being written to replace the MPI function libraries. The parallel program already coded will need no changes to be added to a BONeS simulation. The new MPI functions provided by this library act as the application layer for the lower layers of the BONeS model already developed. The role of the library as an interface between regular MPI parallel programs and BONeS is shown in Figure 6.3.1. Furthermore, the libraries contain probes which collect detailed information on each communication, including number of bytes sent and the time step at which the communication occurred. While running times of parallel programs were useful in the coding stage of development to determine which decomposition methods were better and by how much, at this point in the development track, the goals are different. The running times over a SPARCstation testbed are no longer a concern because any running times do not reflect the running times on the sonar array. Though the BONeS simulation does not occur in real time, the simulator keeps track of time steps. Knowing the step-by-step information from the probes in the BONeS interface library, only the values for latency and throughput of the physical sonar array will be needed to construct a true-to-life representation of the timing of the final array.

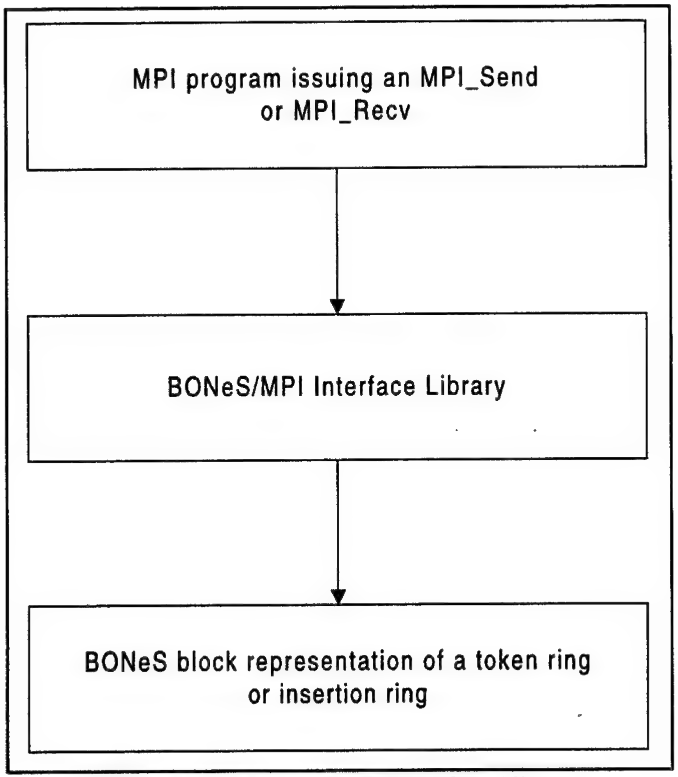

MPI program issuing an MPI_Send
or MPI_Recv
BONeS/MPI Interface Library
BONeS block representation of a token ring
or insertion ring

Figure 6.3.1 : BONeS/MPI Interface Library
This block diagram shows the layers for the BONeS/MPI Interface Library.

The BONeS/MPI interface will go through two phases, shown in Figure 6.3.2. The first will simulate the parallel program no differently than BONeS usually carries out a simulation. BONeS will use multiple workstations on a network only for different iterations in its simulation, but it will not split up simulating work for a single iteration between workstations. Using this method, the BONeS simulation of the parallel program over a network architecture will not be done in parallel. Instead, the entire program for a single iteration will be simulated on one machine, and the probes will keep track of the time steps of important events. The second incarnation of the interface library will force the BONeS simulation to run over multiple workstations, splitting up the array nodes accordingly. This can be done since BONeS provides the code it uses to run the simulation, so parallelization of the BONeS simulation is possible. To parallelize BONeS, the real MPI will be used, as opposed to the library functions for the BONeS/MPI interface as discussed above.

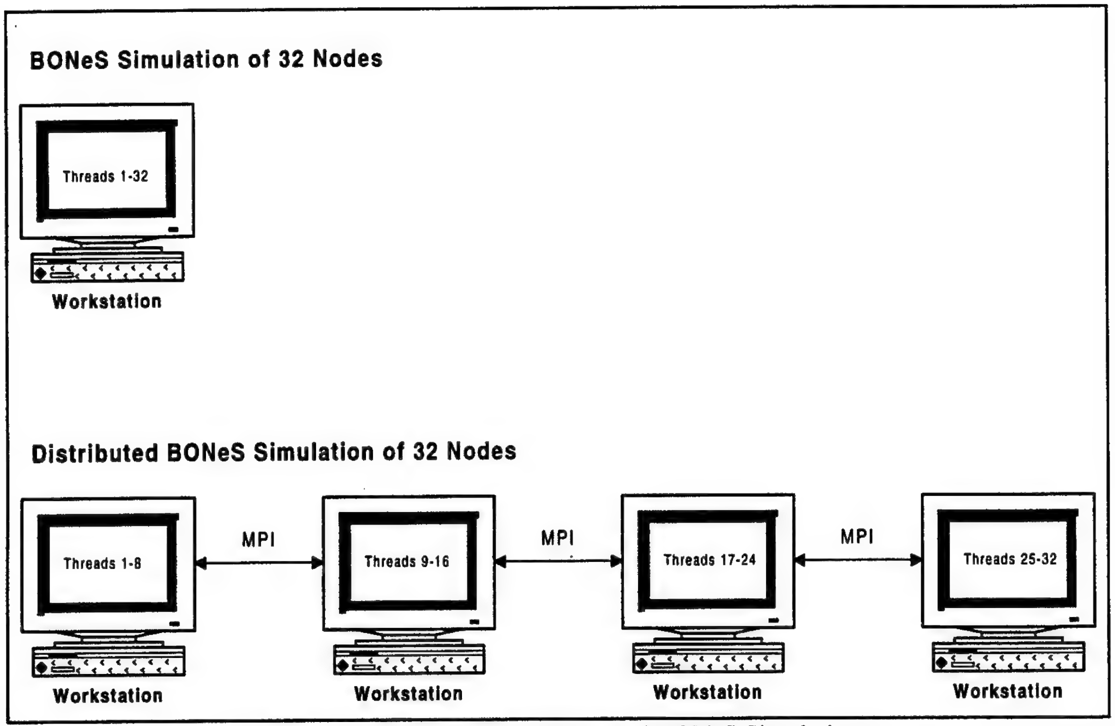


Figure 6.3.2 : Regular and Distributed BONeS Simulations .

The top half of this picture shows how BONeS will simulate 32 nodes in the first implementation. The bottom half shows the same simulation with distributed BONeS.

Up to this point, a number of layers have been put on top of each other. A user places code fragments into a parallel program framework, which is interfaced to BONeS, which is distributed over several workstations. Figure 6.3.3 illustrates how all the layers of a simulation fit together.

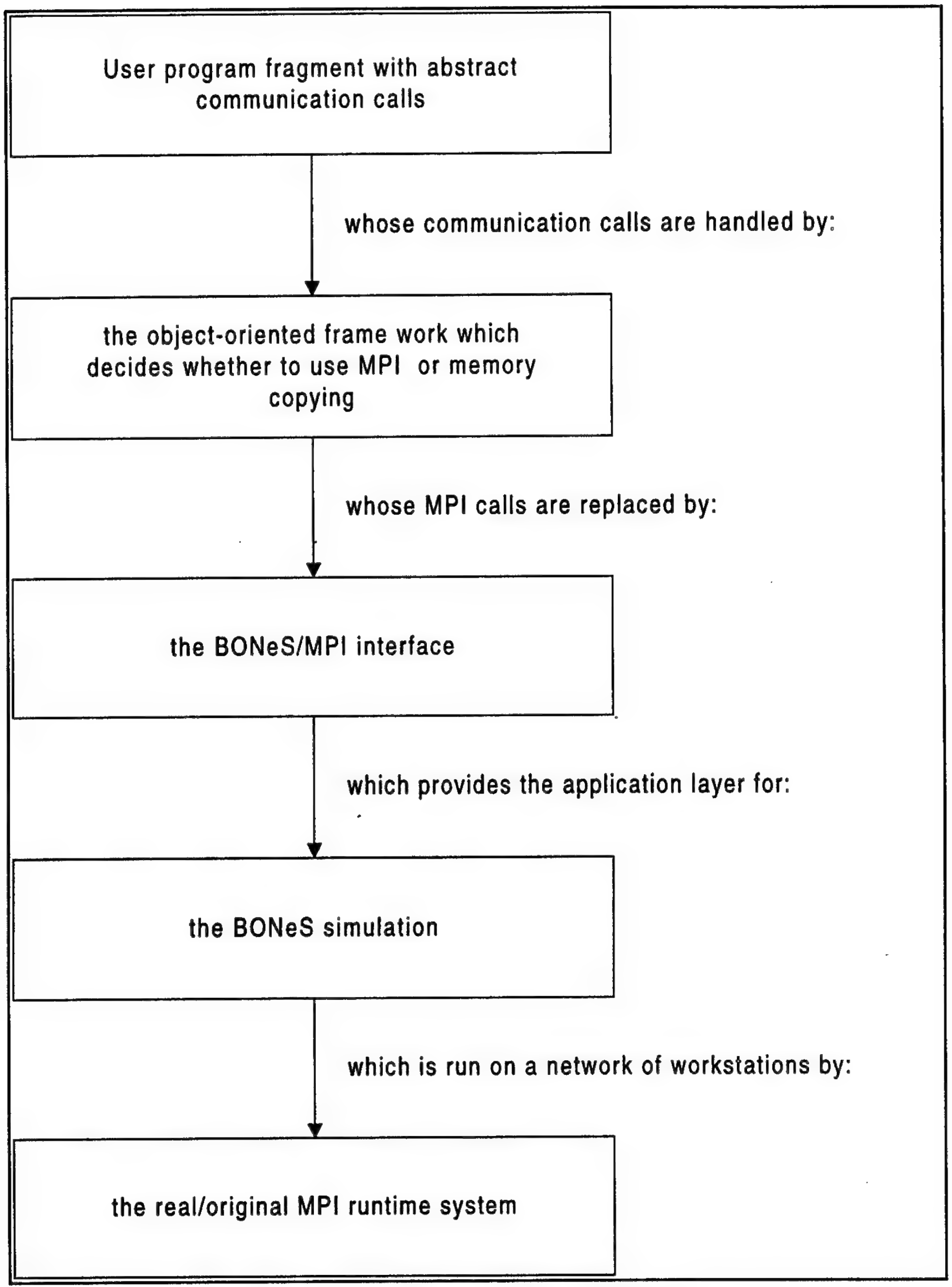

User program fragment with abstract communication calls
whose communication calls are handled by:
the object-oriented frame work which decides whether to use MPI or memory copying
whose MPI calls are replaced by:
the BONeS/MPI interface
which provides the application layer for:
the BONeS simulation
which is run on a network of workstations by:
the real/original MPI runtime system

Figure 6.3.3 : Simulation Layer Hierarchy

This figure depicts the layers of the complete algorithm development track, from parallel coding to parallel BONeS simulation.

6.4 GUI for Distributed BONeS/MPI Interface

The fourth and final step in the development process is the use of a graphical user interface running over the distributed BONeS simulation to report simulation information. The information included in this final GUI will be considerably more in depth than the information on timing that the GUI or text-based runtime system currently provides. Because the probes from the BONeS interface provide information on the simulation time step by time step, the final GUI will be able to graphically display the work of the array in an intuitive, step-by-step manner. Along with considerable information on the source, destination, and size of each communication, the GUI will display the link usage graphically. Figure 6.4.1 shows a prototype format for the final GUI.

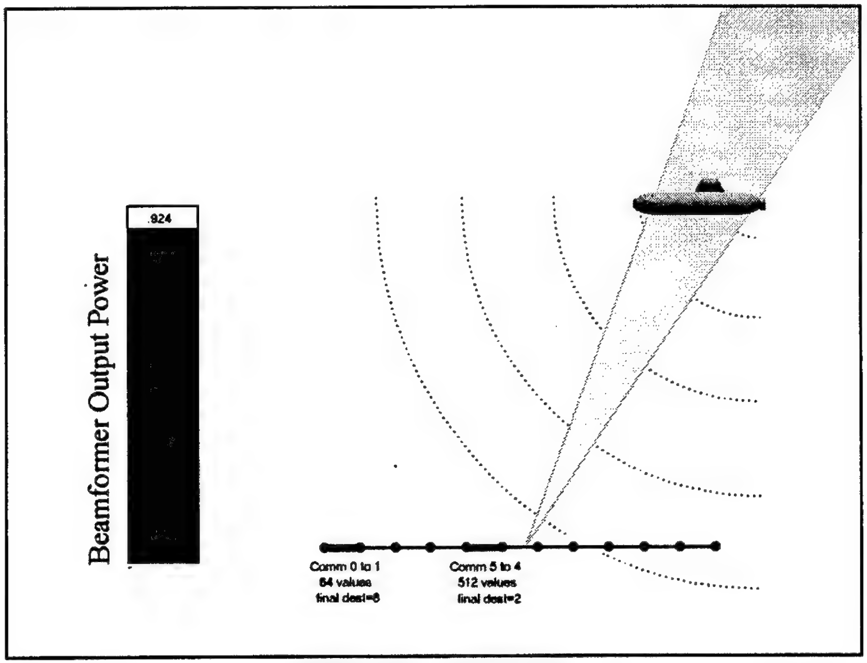


Figure 6.4.1 : Final GUI

The above figure shows a prototype screen shot from the final GUI. Notice the link utilization information on the highlighted links and the beamformer output result number.

More specifically, the final user interface will allow the user to inject faults into the simulation and change parameters for the physical array and incoming source. It will also provide several choices for the algorithm to run and for the architecture over which it runs. The final GUI will also control the size of the BONeS simulation, such as the number of iterations or the number of workstations over which the simulation is distributed. A non-exhaustive list of parameters to be supported is shown in Figure 6.4.2. Many of these parameters have already been implemented in the simpler text-based runtime system and GUI.

Incoming Frequency
Speed of Sound
Source Horizontal Distance
Source Radial Distance
Source Angle
Sampling Frequency
Spherical Spreading
Number of Sensors
Angles per Iteration (Granularity)
Working Node Density (Granularity)
SMP
Sensor Spacing
Interpolation Factor
Beginning Search Angle
Ending Search Angle
Search Angle Increment
Number of Samples
Array Windowing Function
Interconnection Testbed
Thread Level
Commandline
Parameters
Windowing Function Alpha
Number of Iterations
Nodes per Packet (Granularity)
Number of SPARCstations
Print Debug Information
Topology
Network Clock Speed
Output Queue Size
Failed Nodes
Clock Skew Percentage
Fault Injection Timing Factor
Master Timeout Factor
Propagation Delay Time

Figure 6.4.2 : Non-exhaustive List of Parameters

A list of all the parameters the final GUI will control.

With these simulator tools, parallel algorithms can be implemented in a true-to-life software prototype simulation. The results of these simulations can be used to make a correct decision as to the details of the final hardware.

7. Preliminary Software System

7.1 Software Introduction

There are several different programming and testbed implementations that are used throughout the program runs that affect the timings. The differences in timing caused by the different simulation and implementation methods is of interest in determining how efficient the different methods are. Of the different methods, it is the testbed network interconnect and the thread library used which are of the most interest. The optimized programs were run on either an Ethernet testbed, an ATM testbed, or an SCI testbed. Furthermore, some programs used either the HCS threads implementation or the Solaris threads implementation.

Between the different implementations, the execution times can change significantly for the same program. For example, in an extreme case, the SCI version with HCS threads outperforms the Ethernet version with Solaris threads by a factor of 1.9. If not handled correctly, this factor of 1.9 will significantly affect the comparisons made between parallel algorithms. To rectify the problem, all speedup comparisons of parallel algorithms for this project are done with the "like" baseline. Any program running Solaris threads will be compared with a Solaris thread version of the baseline, and any program running HCS threads will be compared with the baseline running HCS threads. The same requirement holds for whether the parallel program is running over Ethernet, ATM, or SCI. With the comparison methods in place, speedup numbers between algorithms will be valid because all implementation-dependent influences will be factored out and the implementation-independent factors become clear.

7.2 Time-Domain Implementation

In Section 5.4, the delay-and-sum with interpolation (DSI) algorithm was decomposed into functional and domain techniques, and it was concluded that the domain decomposition technique provides a better parallel algorithm in that it allows for better fault tolerance, more naturally fits the algorithm, and requires less communication among the processors. This section implements this time-domain technique to quantify the performance of the decomposed algorithm over a distributed array, the speedup of parallelizing this conventional beamforming technique and the performance benefits when executed over different communication networks. In the sonar array, each of these factors must be addressed in order to determine the usefulness of parallel processing on the sonar array. In order to show the performance improvement of the proposed parallel processing techniques, a baseline system was also simulated which implements a passive sonar array with an end processor. The following sections detail the simulation of the DSI algorithm over the baseline and in parallel over a unidirectional topology, a ring topology, and a bidirectional topology.

The results indicated in the following sections represent a preliminary investigation into the parallelizability of time-domain beamforming algorithms. The primary focus of the time-domain research was the application of standard parallel programming techniques to a distributed processing sonar array. Conversely, the optimal execution of the algorithms was the goal of the frequency domain implementations but the results of both endeavors are of equal importance. For this reason, the time-domain implementations use a simple execution and communication model which does not support a complex simulation environment. The drawbacks and limitations of this simulation method prompted the development of the abstract programming framework presented in the previous section. The later implementation of the frequency-domain algorithms prototyped the simulation techniques abstracted in the framework. By using the direct, low-level communication functions available in a workstation cluster, the time-domain simulations quantify the performance of the testbed as a simulation device as well as providing insight into parallel time-domain beamforming algorithms in general.

7.2.1 General Implementation for the Parallel DSI Algorithm

Many factors were considered in the simulation of the sonar array on which the delay-and-sum with interpolation (DSI) algorithm was to be applied, including how to simulate each sonar node, how to simulate each network topology, and how the DSI algorithm would be implemented on the sonar array. To simulate the sonar array, each array node was implemented as a single process on a workstation. As a result, the number of nodes comprising the sonar array was limited to the number of workstations when using MPI for SCI (MPI/SCI) or to the number of processors using MPI for Ethernet or ATM (MPICH). By allowing only a single process per processor, the simulation implementation was much simpler. A multithreaded implementation is also possible but may or may not improve simulation speed since the inter-workstation communication is the major limiting factor. All inter-workstation communication was handled by the MPI implementation specifically designed for the network in use. All timing runs were carried out on a testbed cluster of 8 dual-processor workstations capable of using Ethernet, SCI, and ATM network protocols.

For MPICH on Ethernet and ATM, the algorithm was able to simulate up to 15 array nodes on 8 workstations. The command to run an MPICH simulation is:

```
mpirun -np 13 -machinefile proc20 PUTDSImpi 0
```

where the *-np* flag allows the user to specify the number of processes to run, the *-machinefile* flag specifies the file listing the workstations to be used, *PUTDSImpi* is the MPICH program, and *0* represents the command line argument for the simulated algorithm. The processes were mapped in a round-robin fashion on the workstations. For between three and eight array nodes, each process was mapped directly to a workstation. For more than 8 array nodes, each workstation was assigned another process until the number of sonar nodes was exhausted. In this case, the second processor in the workstations is used in parallel but the network interface is shared between the two processes.

The second MPI implementation was MPI/SCI. The command line to run this implementation was, for example:

cluster PUTDSIsci -scilist 4,8,12,16 0

where *PUTDSIsci* is the MPI executable, *-scilist* is the flag to specify the workstations that were to be used, the comma-separated list of numbers represents the workstation's SCI addresses, and *0* is the command line argument for the simulation. Unlike the MPICH implementation, MPI/SCI limits the number of processes to the number of workstations since the SCI interface could not be shared with this version of MPI. For the SCI simulation, this meant that no more than 8 nodes were simulated. Although the number of workstations was limited, this format ensured that the same communication library were required for all communications between nodes in the sonar array. Because only MPI is used for all communications during the time-domain simulations, the timing results can accurately compare the different computer networks used for simulation. This is in contrast to the frequency-domain simulations in which the communications between threads on the same machine uses a local memory copying technique and therefore introduce much different performance characteristics.

The next consideration was how to simulate the network topologies on a cluster of workstations. This was done by defining and enforcing proper inter-node communication rules. For example, the array nodes in a unidirectional topology were limited to communicating only with their immediate downstream neighbor. In the ring topology the most downstream node is also capable on sending data to the most upstream node. In the bidirectional array, each sonar array node is capable of directly communicating with its immediate neighbor. Using MPI, the inter-node communications were forced to pass through any intermediate nodes in the simulated architecture instead of directly communicating with the destination node. Figure 7.2.1 shows an example of this constrained communication to simulate a network architecture.

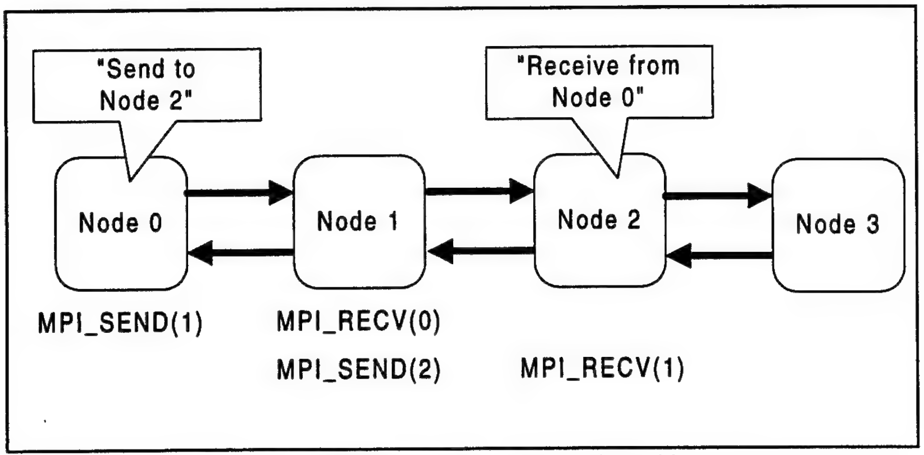


Figure 7.2.1 : MPI Simulation of Network Architectures

This figure illustrates how MPI communications are constrained to use only those types of communications that are available in the target network architecture.

7.2.2 Sequential DSI Beamformer

One of the first steps in the development of the parallel delay-and-sum with interpolation beamformer was to implement the algorithm as a sequential program. A program was written to perform delay-and-sum beamforming using the standard C language and designed to execute on a single workstation. The program served as a benchmark for the basic computational requirements of the algorithm to be parallelized. Also, the results of the different parallel implementations were compared against the sequential results to verify correctness. Figure 7.2.2 shows the flowchart for the sequential program. It is this program which was manipulated to create the baseline and the various parallel programs.

The program starts out by reading a file written by the simulator user interface that lists all the parameters needed by the program. This includes number of array nodes, number of samples per beam power calculation, interpolation factor, sampling frequency, source frequency, source position, user mode, and type of beamforming. The user mode indicates whether the program is being run for timing purposes or for algorithm debugging. The beamforming type indicates whether the beam pattern or the steered response is being considered. For this research, the beam pattern was chosen as the algorithm to parallelize since it requires more calculations. The incoming signal must be calculated each time a new direction is indicated. The purely sequential program simply represents a starting point from which the simulations are based. It is this algorithm that is decomposed and divided among the workstations.

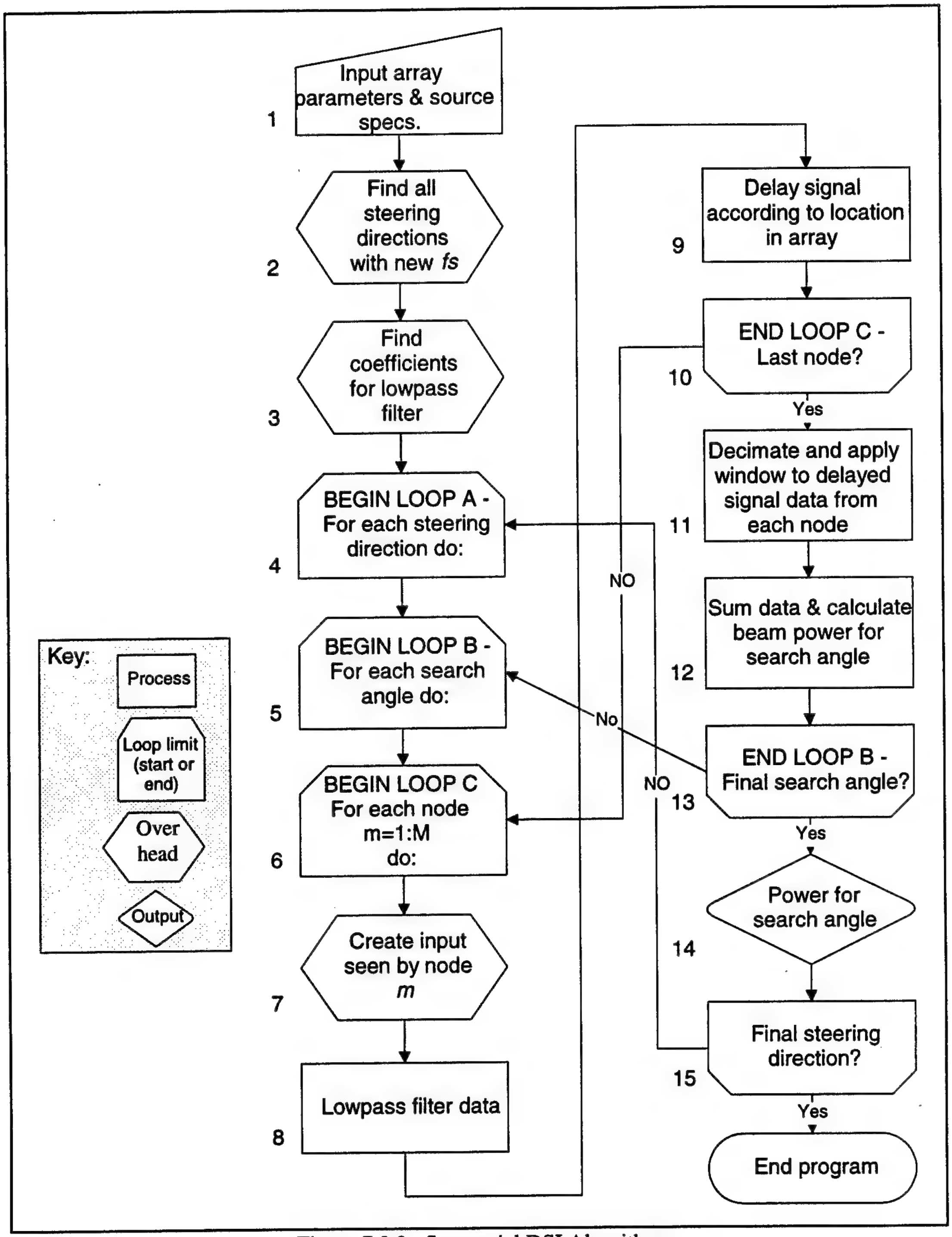

Input array parameters & source specs.
1
Find all steering directions with new *fs*
2
Find coefficients for lowpass filter
3
BEGIN LOOP A - For each steering direction do:
4
BEGIN LOOP B - For each search angle do:
5
BEGIN LOOP C For each node m=1:M do:
6
Create input seen by node *m*
7
Lowpass filter data
8
Delay signal according to location in array
9
END LOOP C - Last node?
10
Yes
Decimate and apply window to delayed signal data from each node
11
NO
Sum data & calculate beam power for search angle
12
No
END LOOP B - Final search angle?
NO
13
Yes
Power for search angle
14
Final steering direction?
15
Yes
End program
Key:
Process
Loop limit (start or end)
Over head
Output

Figure 7.2.2 : Sequential DSI Algorithm

This flow chart shows the execution of the sequential delay-and-sum with interpolation algorithm.

7.2.3 Baseline Specification: Sequential Unidirectional DSI Beamformer

Without a baseline to compare with the parallel programs, the results from the proposed methods would have little meaning. Therefore, it was important to define a baseline similar to a standard sonar array. The array configuration used as the baseline for comparisons is a number of nodes linearly connected together that simply collect data and send that data to the next immediate node. The data is sent via the boxcar network as described in Section 4.5. All data is eventually gathered by a single end processor where it is processed. For the DSI simulations, each sonar node in the array was executed as a separate process. The inter-nodal communication was implemented exclusively using MPI. Each process was responsible for reading the incoming boxcar, calculating the data for the incoming source, inserting this data into the boxcar, and sending the boxcar to the next node. Finally, the boxcar packet arrives at the end processor, where the sequential program on a single workstation is executed on the data. This baseline provides the point of comparison for all proposed parallelization of the array calculations.

7.2.4 Basis for Comparison

The baseline simulator introduces simple communication delays to the purely sequential delay-and-sum beamformer to model a unidirectional linear array, which is the baseline for all architectural comparisons. The added communication delays make the baseline simulations a better basis for comparison (hence the nomenclature) against the parallelized versions of the beamforming algorithms. By comparing against this baseline, the computational attributes of the different parallel implementations can be compared, thereby relying less on the performance of the network used during the simulations.

As previously stated, all speedup numbers calculated for the parallel programs in this section were made by comparison with the "like" interconnect. For example, all ATM parallel programs are compared with the ATM baseline and all SCI parallel programs are compared with the SCI baseline.

It is also important to recall that the variable *L* shown in all of the following charts represents the interpolation factor. Larger values for the interpolation factor mean a larger problem size.

7.2.5 Parallel Unidirectional DSI Beamformer

Three main parallel programs were implemented. The first of these was the parallel unidirectional time-domain program (PUT). In this algorithm, each node was responsible for generating its own data. Figure 5.4.3 illustrated how this algorithm was implemented. That figure is presented again below as Figure 7.2.3. Each block across a single row represents a process. The last node on the array, the highest ranked node, calculates its data and creates the vector that will be sent along the network by delaying its data the appropriate amount and sending it on. The next node then receives that data with an *MPI_Recv*, delays and sums its data with the vector being passed along, then sends it to the next node down the line. Eventually this reaches the first node, which then performs the beam power processing to obtain the beam pattern. Then the process begins again. The disadvantage of this technique is that even though some of the other nodes are taking the burden off the end processor, it is still this end processor that performs the most calculations and has the potential for causing a bottleneck. On the other hand, the sampling frequency of the array may not be fast enough where that becomes a problem. Besides the parallel processing, a major difference between this configuration and the baseline is that should the end processor fail in PUT, the next node can take over its responsibilities such that it now becomes the main node. All nodes have the computational capacity to assume this responsibility. If this happened in the baseline, the array would be rendered useless.

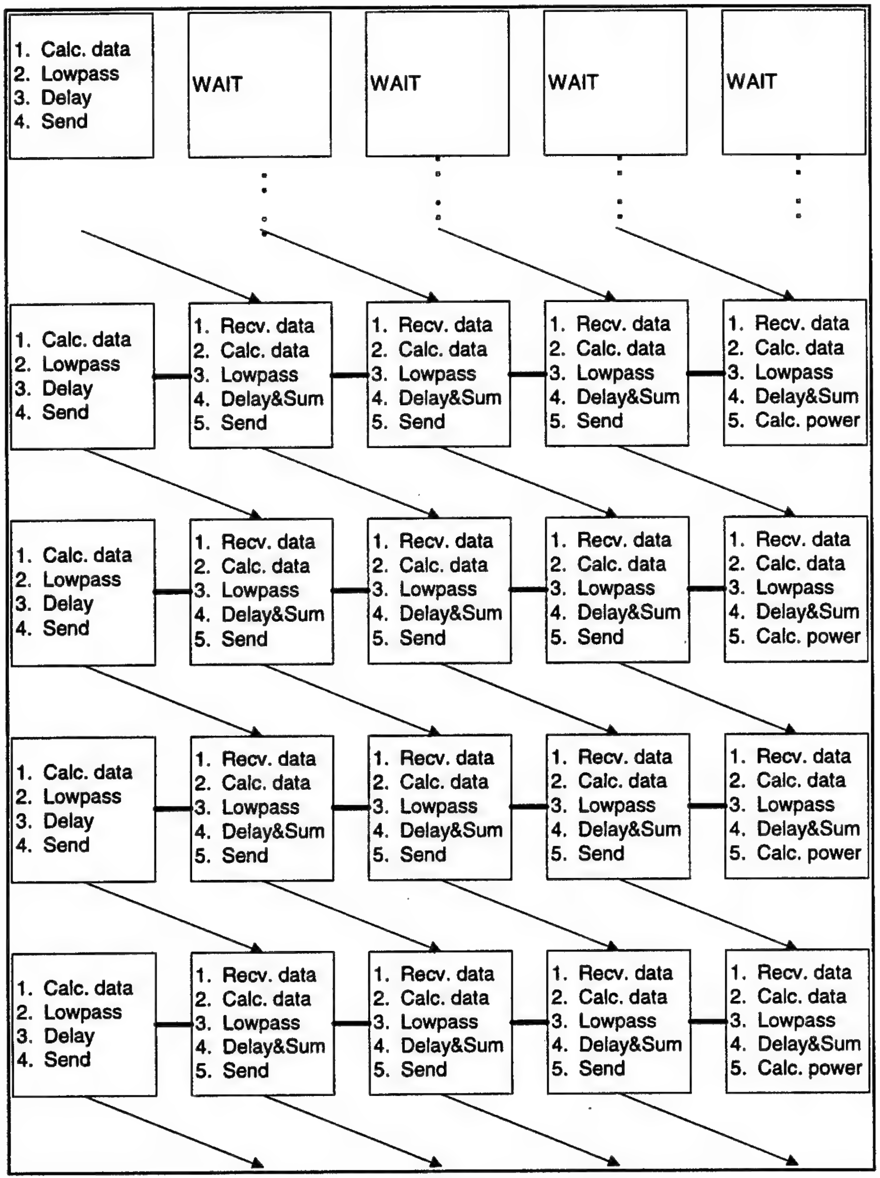


Figure 7.2.3 : Unidirectional DSI

As can be seen by the uniform direction of the arrows, this flow chart depicts the unidirectional delay and sum with interpolation.

When comparing PUT with the baseline, there is a definite opportunity to achieve speedup. The question, however, is how much speedup will be attained. The decomposition methods employed spread the addition of the delayed data samples out over the nodes, thus lessening the amount of communication required by each node. To elaborate, the packet size in the baseline equals 64 samples times the number of nodes. In PUT, the communication remains constant since each node sends the same 64-sample packet.

This decrease in communication and decrease in amount of calculations performed by the head processor means that parallel speedup can be attained. Figures 7.2.4 and 7.2.5 illustrate these speedups.

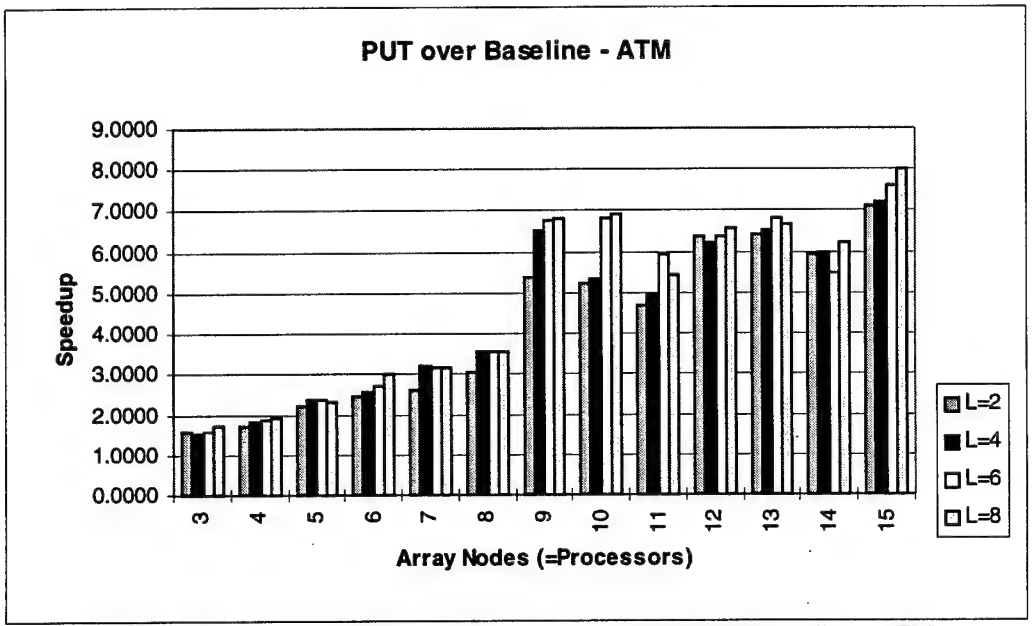


Figure 7.2.4 : Speedup of Parallel Unidirectional DSI over the Baseline - ATM

This figure depicts the speedup of the parallel unidirectional program over the baseline when using ATM for both. The different values for *L* in the legend show different interpolation factors.

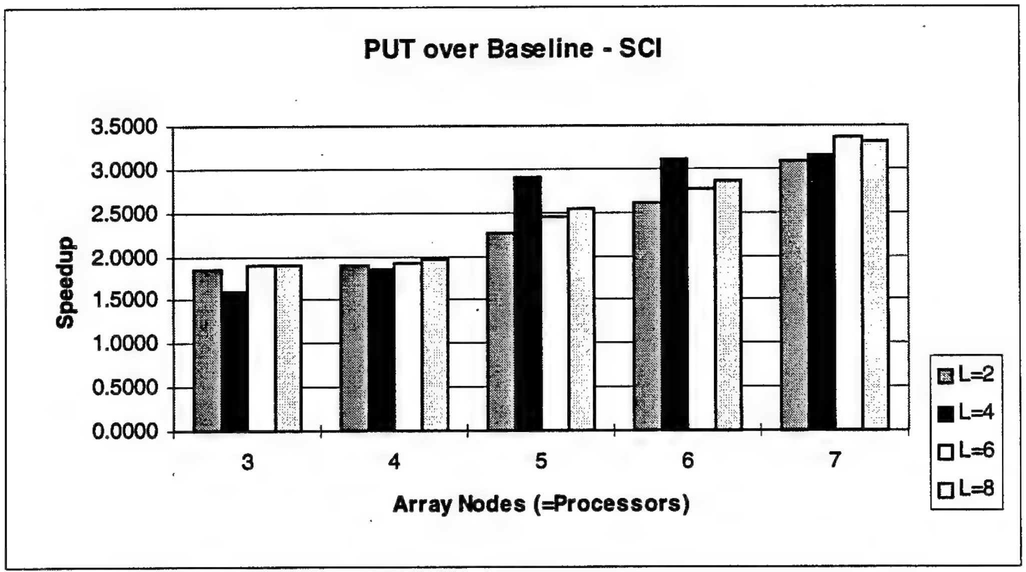


Figure 7.2.5 : Speedup of Parallel Unidirectional DSI over the Baseline - SCI

This figure depicts the speedup of the parallel unidirectional program when using SCI over the baseline when using SCI.

As expected, the speedup rises as the number of nodes increases. In addition, the higher the interpolation factor, the better the speedup for most cases. This is a result of the increase in the number of calculations required for the interpolation and the ability of the parallel algorithm to spread out the increased computations over the various nodes.

7.2.6 Parallel Ring DSI Beamformer

The next configuration, parallel ring time-domain (PRT), is based on a revolving head processor. Other than this factor, the ring is very similar to PUT because it is unidirectional and each node is sending to a specific end processor. By revolving the end processor, the bottleneck mentioned in the PUT algorithm can be relieved.

When comparing the baseline against the ring topology, the speedup once again is not expected to reach that of linear because the communication and topology setup still poses overhead that is inevitable. Figures 7.2.6 and 7.2.7 show the speedup curves for the baseline versus parallel ring time-domain DSI.

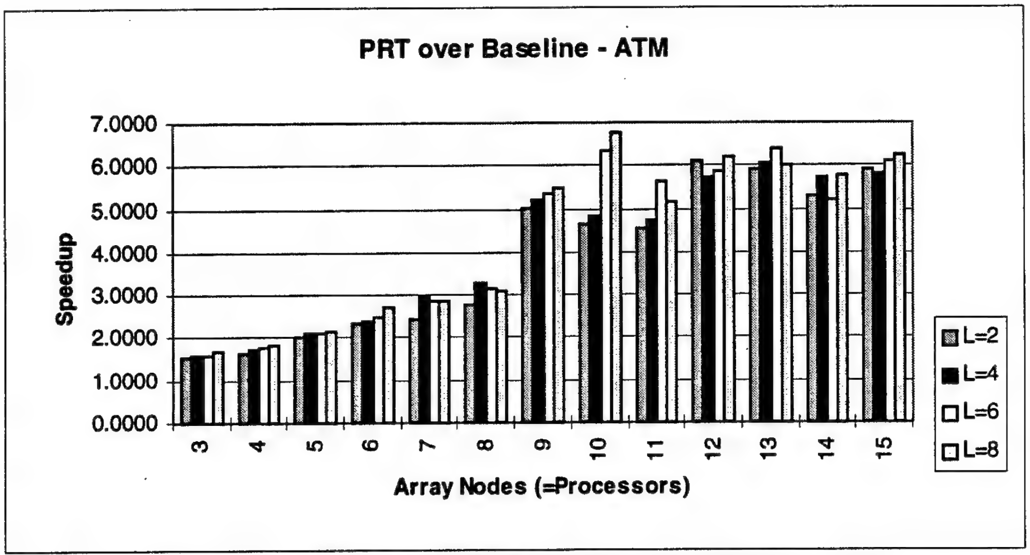


Figure 7.2.6 : Speedup of Parallel Ring DSI over the Baseline - ATM

This figure depicts the speedup of the parallel ring program using ATM over the baseline using ATM.

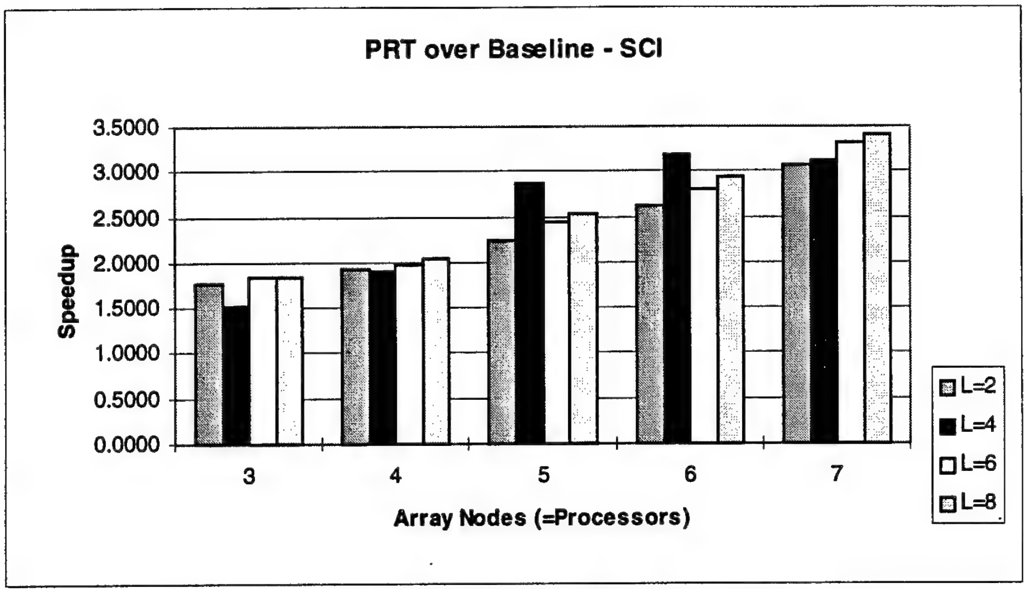


Figure 7.2.7 : Speedup of Parallel Ring DSI over the Baseline - SCI

This figure depicts the speedup of the parallel ring program when using SCI over the baseline when using SCI.

Generally, the speedup increases as the number of nodes is increased. In the ATM runs, most of the execution times for greater than 8 nodes had a large variance. This could be a result of a number of factors, mainly the simulation method. By assigning two processes to some of the workstations, not only do the processors share the execution of the processes, but the bandwidth of the network as seen by each process also effectively decreases because it is now being shared by two array nodes instead of one.

7.2.7 Parallel Bidirectional DSI Beamformer

Finally, the PBT algorithm was implemented as described in Figure 5.4.4. That figure is presented again below as Figure 7.2.8.

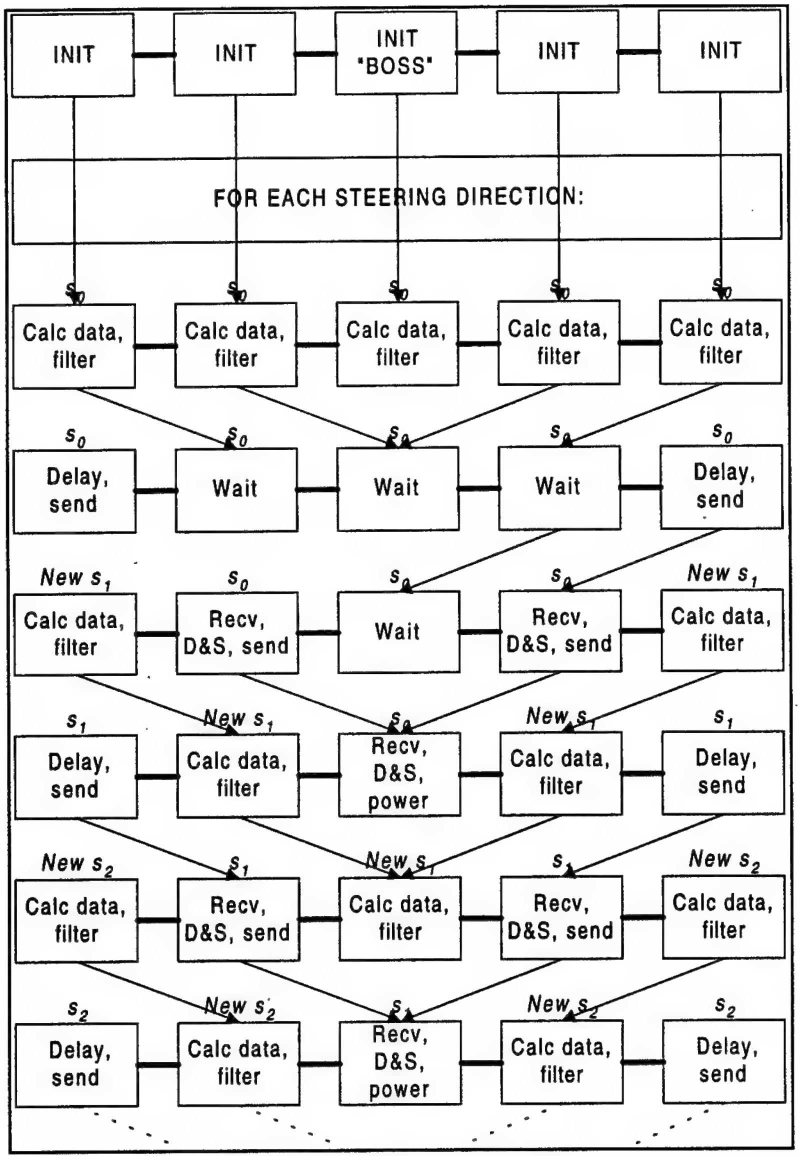


Figure 7.2.8 : Bidirectional DSI

This figure is the same as Figure 5.4.4 and depicts the bidirectional delay and sum with interpolation.

The bidirectional algorithm is significantly different from the ring and unidirectional algorithms. Rather than having a floating front-end, as in the ring case, the bidirectional array uses the middle node as the boss (the effective front-end). Timings and speedup numbers were taken in the same manner as the ring program and are presented in Figures 7.2.9 and 7.2.10.

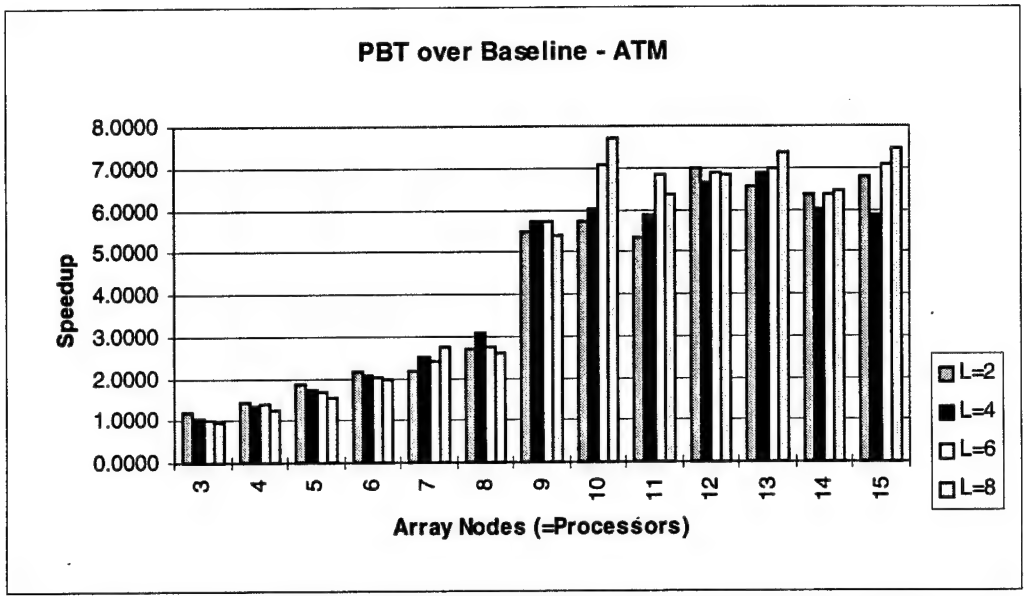


Figure 7.2.9 : Speedup of Parallel Bidirectional DSI over the Baseline - ATM

This figure depicts the speedup of the parallel bidirectional program when using ATM over the baseline when using ATM.

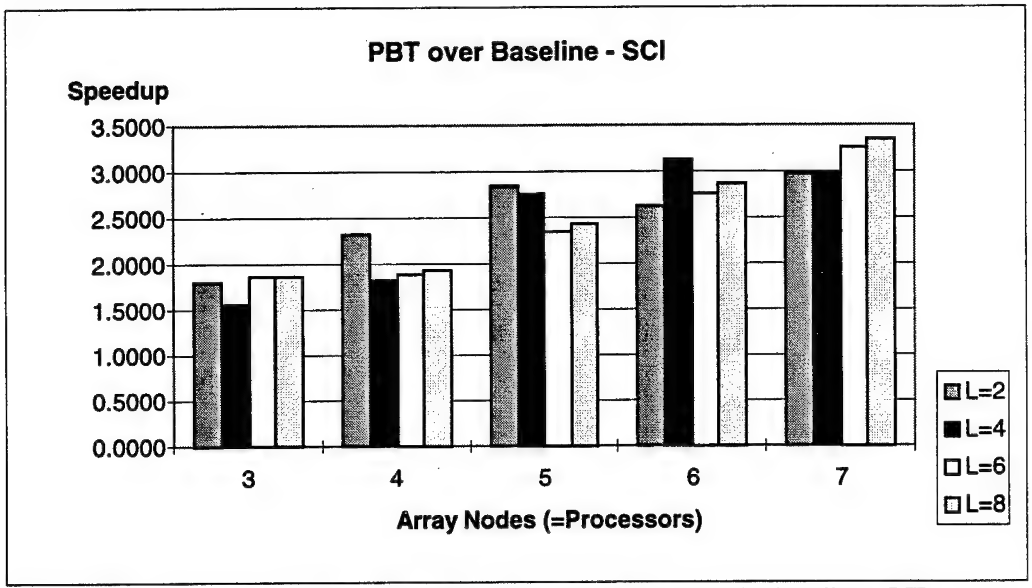


Figure 7.2.10 : Speedup of Parallel Bidirectional DSI over the Baseline - SCI

This figure depicts the speedup of the parallel bidirectional program when using SCI over the baseline when using SCI.

These charts show the similar performance of the PBT algorithm and the PRT algorithm. The speedup of PBT peaks to about 3 for 7 processors over both SCI and ATM. The ATM runs maintained similar performance to the PRT as more nodes were added.

7.2.8 Parallel Program Comparison

The previous discussions emphasized the speedups of the various programs over the baseline. Rather than observing their speedups versus the baseline, this section will present the algorithms side-by-side in order to make comparisons between them. The speedups of the parallel versions of all the programs for an interpolation value of 2 are placed next to each other in Figure 7.2.11. The same programs for an interpolation value of 8 are shown in Figure 7.2.12. The ATM and SCI programs are shown next to each other even though the SCI programs could not be run with more than 8 processors. This allows comparison between the programs over the different interconnects (for 8 or fewer nodes) as well as comparison between the programs for ATM on larger problem sizes.

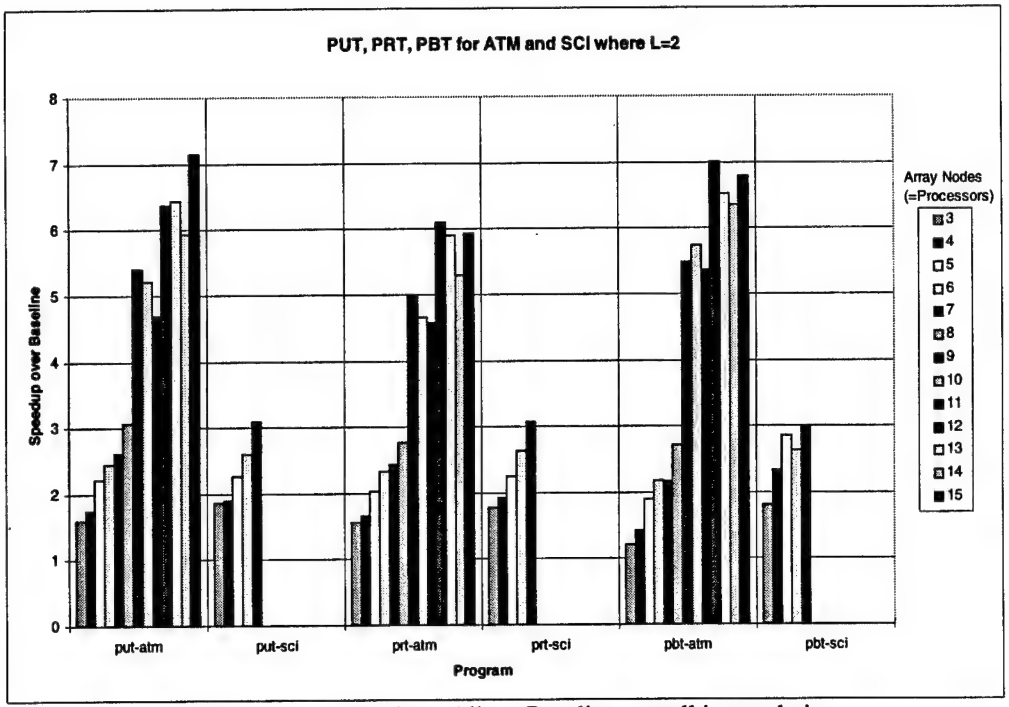


Figure 7.2.11 : Speedup - All vs. Baseline - small interpolation
ATM and SCI speedups of all the parallel programs over the Baseline (with "like" interconnect) for interpolation=2.

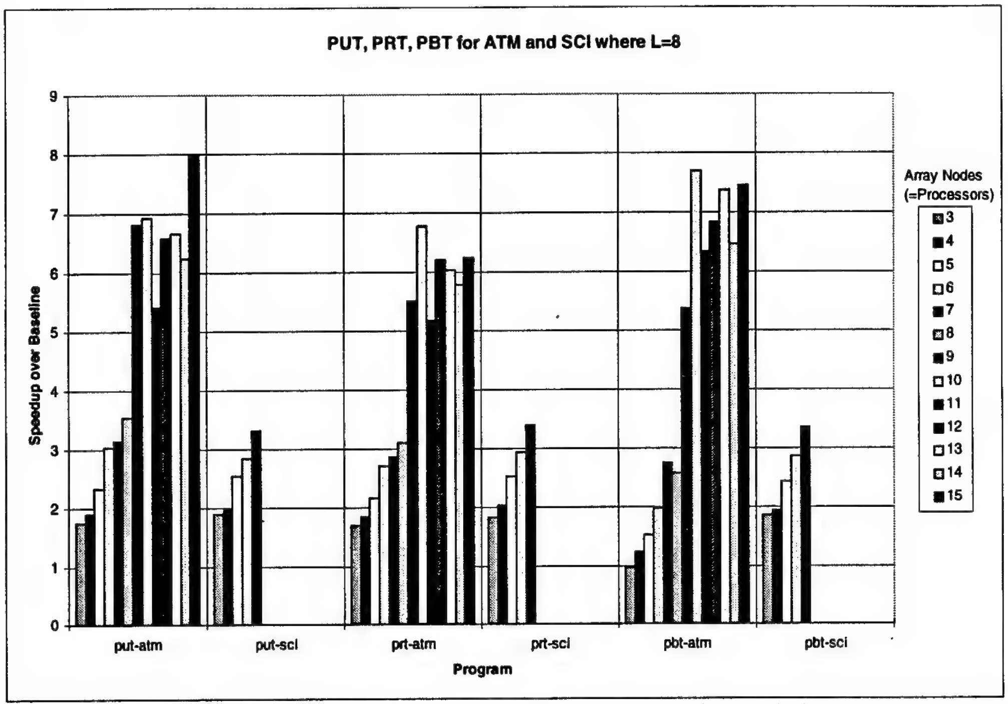


Figure 7.2.12 : Speedup - All vs. Baseline - large interpolation

ATM and SCI speedups of all the parallel programs over the Baseline (with "like" interconnect) for interpolation=8.

As can be seen in the graphs, all the algorithms scale fairly well. As more processors are added and the problem size increases, the speedup increases. This speedup is not linear with the number of processors in that the speedup did not reach 15 for 15 processors, but the programs do perform considerably better than the baseline. Also, it is interesting to note that the unidirectional, ring, and bidirectional time-domain programs perform at the same level. No one algorithm stands out as a preferred method. This is somewhat surprising when considering that the unidirectional program is placed under the restriction that the data cannot be passed up to nodes at the most upstream side of the array. Without being able to communicate to the upstream processors, the possible degree of parallelism is severely limited because those upstream processors will be incapable of accepting shares of the work. The programs which took advantage of a fully-connected, bounded-degree network (PBT and PRT) are capable of a higher degree of parallelism due to the fact that all nodes can be fully utilized because no node is trapped with a one-way communication.

7.2.9 Parallel Speedup Over Sequential

The previous discussion concentrated on the results and speedups obtained versus the baseline. This comparison is the kind of most importance to this project and to the field of sonar array processing. However, it is also important to make comparisons between these parallel programs and the purely sequential program. This type of comparison is geared toward studies in parallel computing and computer engineering in general. The speedup over the purely sequential program may also be of interest in off-line beamforming, where data is collected but processed at a later time by standard workstations above water.

The speedups over the pure sequential program for an interpolation value of 2 are shown in Figure 7.2.13. On the horizontal axis, each group shows the values for a particular program, and bars within a group show how many processors were used to obtain the value in that bar.

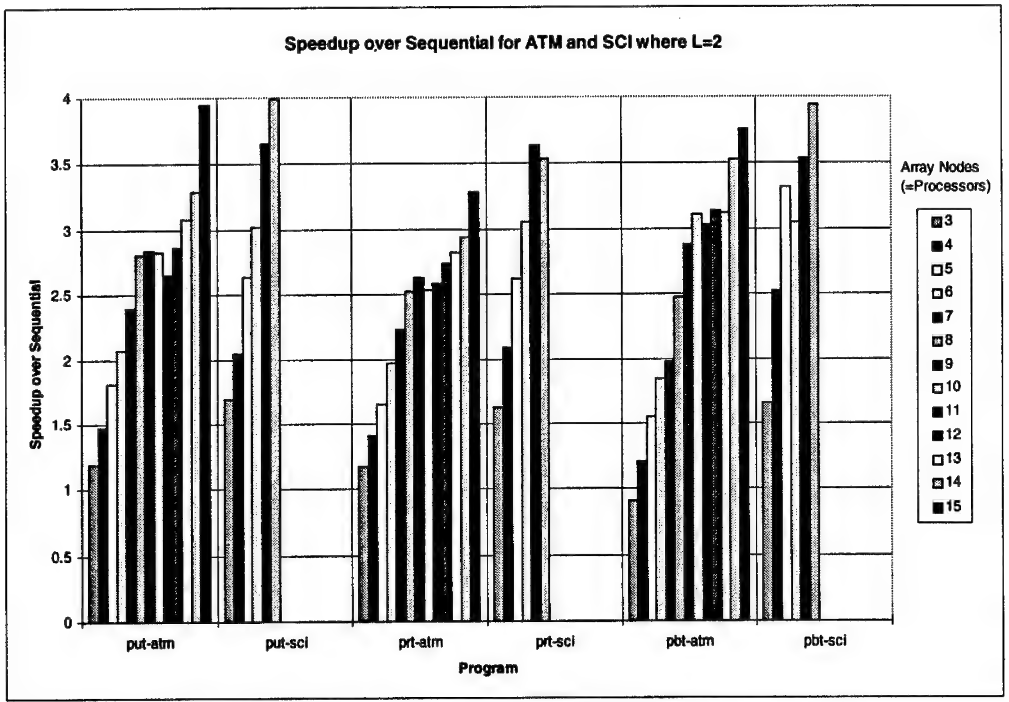


Figure 7.2.13 : Speedup vs. Pure Sequential - small interpolation

Speedups of all parallel programs versus the purely sequential program. Each bar group represents a program. Each bar within a group indicates the number of array nodes, and hence the number of processors used.

Looking at the speedup versus the pure sequential algorithm, it is clear that the parallel programs are still close to performing the same. Just as in the case of comparing to the baseline, no program stands out as a better performer, although the effect of the interconnect becomes more clear.

The same situation holds true for a large interpolation value. The speedups for the case when the interpolation is 8 are shown in Figure 7.2.14. These values show that the algorithms perform acceptably as the problem size is increased.

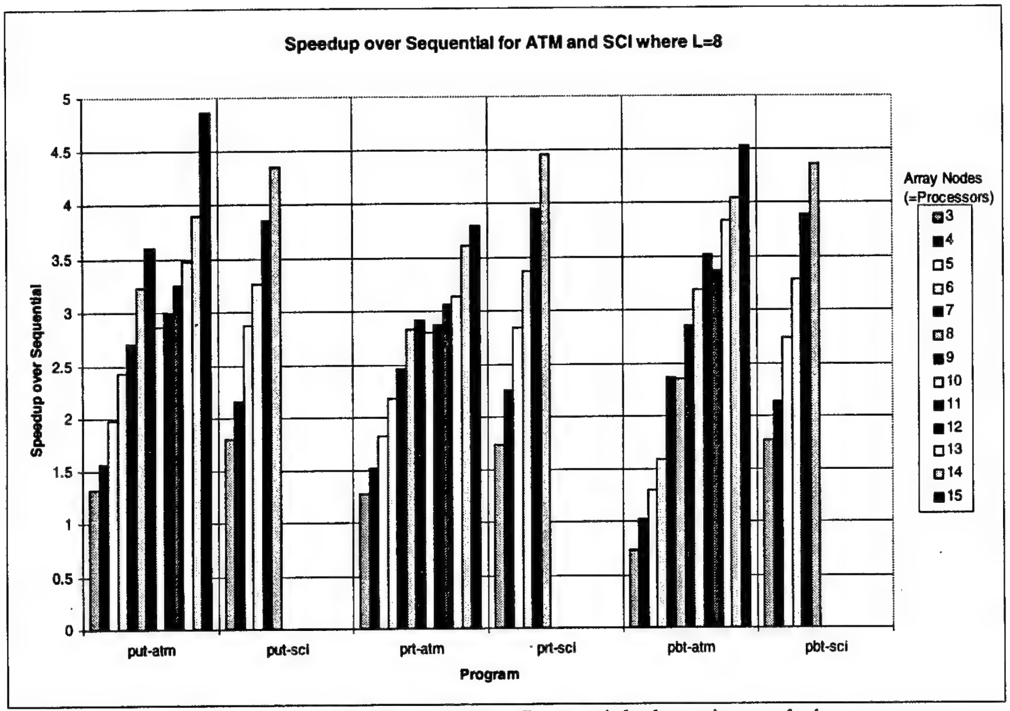


Figure 7.2.14 : Speedup vs. Pure Sequential - large interpolation

Speedups of all parallel programs versus the purely sequential program for interpolation=8.

By looking only at the values for eight or fewer nodes, comparisons can be made between the ATM and SCI versions of the programs. In all cases, the SCI version performs better than the ATM version. For example, for 8 nodes, PUT performs approximately 35% better when running over SCI. For PRF, this number is about 55%; and for PBF, it is 87%.

* * *

Since more time was taken in the study and decomposition of the time-domain algorithms, phenomenal speedups were not achieved in these preliminary implementations. It was shown that time-domain algorithms can be parallelized and that speedup can even be achieved using simplified simulation techniques. The lack of near-linear speedup of the delay-and-sum algorithm implemented does not, however, imply that linear speedup cannot be achieved. On the contrary, the experience gained implementing this algorithm pointed out two areas of research that needed further examination: optimal simulation techniques and rapid prototyping of beamforming algorithms. Optimized simulation techniques were addressed in the frequency-domain implementation where a multithreaded simulation engine replaced the simple execution model. The object-oriented programming framework that was developed allows rapid prototyping of algorithms over arbitrary processing arrays by abstracting the communications between nodes of an distributed array. The results here also prove that simple, coarse-grain simulations of complex systems cannot offer the precision and granularity necessary to draw concrete conclusions.

7.3 Frequency-Domain Implementation

Basic frequency-domain beamforming algorithms were also designed and implemented. Specifically, the algorithm used was a standard radix-2 Fast Fourier Transform beamformer. As stated in Chapter 5, the FFT beamforming algorithm was used as the representative algorithm for several simulation methods in the implementation. The two most important simulation techniques are multithreading and abstract communication.

7.3.1 General Implementation for the FFT Algorithms

The algorithm was implemented by creating as many threads on the multiprocessor testbed as the number of desired array nodes to simulate. The threads are split evenly across the workstations. For example, 64 array nodes may be simulated by placing 8 threads on each of 8 workstations or by placing 16 threads on each of 4 workstations. A main thread, called the parent, is in charge of setting up the simulation, which not only includes creation of the threads, but also includes simulation of the audio transceiver for the creation of the data the threads will use. The parent thread reads from the configuration file or the runtime system to get the parameters it uses in preparing the incoming data. Timing the execution is also a responsibility of the parent thread, which it does by blocking the threads' execution until it starts the timer and waiting for all of the threads to finish before stopping the timer.

Once the threads have been created and have been unblocked by the parent thread, they begin execution of the beamforming function. It is important to note that all threads execute the same function, distinguished only by the passed input parameter indicating the thread's position in the array. The executed function hides the fact that each thread is not the same; nodes adjacent in the sonar array may sometimes be on the same workstation in the simulation and may sometimes be located on different workstations. In the case of intra-workstation threads, any communication done between them is accomplished by two semaphore exchanges and a memory copy. The receiving thread will post a "ready-to-receive" semaphore when it is expecting another thread to send it something, and will then wait. The sending thread waits for the "ready-to-receive" semaphore and proceeds with the memory copy once it has found that semaphore. After finishing the copying, the sending thread posts a "send-is-done" semaphore, which is the indication to the receiving thread that it may continue execution knowing the communication was successfully completed. This process is shown in Figure 7.3.1. For the case of inter-workstation threads, the Message-Passing Interface must be used to get information from one to the other. The sender calls *MPI_Send*, which returns only after the information is successfully communicated or at least successfully stored in the receiver's communication buffer. The receiver calls *MPI_Recv*, which blocks the process until the information is safely stored in the receiver's memory space.

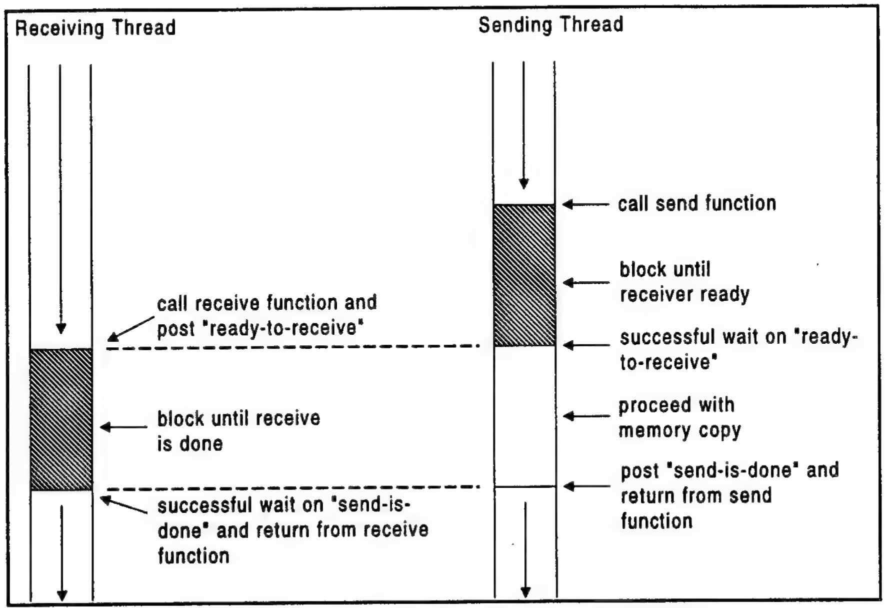

Receiving Thread
Sending Thread
call send function
block until receiver ready
call receive function and post "ready-to-receive"
successful wait on "ready-to-receive"
block until receive is done
proceed with memory copy
post "send-is-done" and return from send function
successful wait on "send-is-done" and return from receive function

Figure 7.3.1 : Intra-Workstation Communication

This figure illustrates the threads of execution when semaphores are used for communication. Shaded areas indicate times the threads are blocked.

A major achievement of the multithreaded simulation was the abstraction of communication so that the threads simply execute an opaque send function or receive function. The simulated threads need not be concerned about whether the thread with which it wants to communicate is on the same workstation or a different workstation. Furthermore, threads in an intra-workstation communication do not worry about posting and waiting for the various necessary semaphores, and the threads in an inter-workstation communication do not worry about passing all the required parameters to the MPI function. With this communication abstraction, the threads simply call a generic send or receive function, specifying only the data size, the data location, and the source thread, and the destination thread. There is an underlying layer which converts the abstract communication call into the necessary code. For intra-workstation communication, the generic communication function call handles all the semaphore and memory-copying operations transparently to the thread or the user. For inter-workstation communication, the MPI implementations used encountered problems when various threads called MPI routines. To correct this, the parent process converts the abstract communication calls of all the threads for that workstation into the MPI calls. The parent process releases the threads when the MPI calls return.

Another method used in the FFT algorithm implementations is a rudimentary network simulation. This is accomplished by simulating the complexity of a communication over different network architectures and is implemented using the communication calls over no more than one link at a time. For example, the implementation for a register-insertion ring architecture is simulated by forcing each node to be able to only send to the thread immediately downstream rather than send to any thread in the array.

There are a number of motivations for implementing an algorithm with such methods. First, the use of threading requires the programmer to think in terms of the code the final project will be running, but the programmer is not confined by the number of machines used for simulation. The backbone of the frequency-domain implementations is the assignment of each array node to a thread, which makes possible the simulation of more array nodes than there are workstations on the available testbeds. Furthermore, by the definition of a thread, the code each thread executes is independent of the code of other threads and the threads execute logically in parallel. These facts mean that the function each thread runs in the simulation is self-contained code for that array node, and no logical changes need to be made in the code when it is migrated from the multithreaded simulation to the actual sonar array.

Second, despite simulating more array nodes than there exist testbed processors, the parallel multithreaded code is still useful for taking meaningful timing measurements. The reason for this is that the communication calls include approximately the same overhead whether using MPI or using semaphores and memory copying. The results will show that the timings for the baseline for a given number of array nodes is approximately the same no matter the number of different workstations running the program. Spreading threads out over more processors does not change the execution time very much, even though the ratio of the number of intra-workstation communications to the number of inter-workstation communications has changed. The explanation for this is that the intra-workstation communication between threads is only slightly faster than inter-workstation communication. It will be seen that the multithreaded method is valid by the consistency of the baseline timings.

The third motivation for using the multithreading involves the rudimentary network simulation. It is completely possible for any thread to send to any other thread with one communication call by way of a completely-connected point-to-point network; however, it is desired to simulate more restrictive network interconnection schemes, such as the ring and the bidirectional linear array. As mentioned before, this is accomplished by forcing communication to proceed link by link in only the allowed directions. The motivation for doing it this way is that a particular algorithm can be run over more than one architecture to determine the feasibility of using that algorithm over that architecture. Though not exact in the timing simulation for a particular architecture, this method is useful for general comparisons between different implementations of the same algorithm.

The final motivation for using this method of simulation is that consistency in the simulation overhead across multiple algorithms assures valid comparisons. Not only can an algorithm be compared over different architectures, as discussed above, but different algorithms can be compared. This is possible because all frequency-domain algorithms implemented here, including the baseline, were coded with the same threading structure. As long as the there are the same number of threads per workstation, multiple algorithms can be compared since the thread overhead can be factored out. The parallel algorithms can be compared with the baseline strictly on the basis of the quality of the algorithms.

The first step in any algorithm parallelization project is to implement the algorithm sequentially. This was done with the SEQFFT program. It serves as a benchmark of the basic computational requirements of the algorithm to be parallelized. The program is single-threaded, so it runs on a single processor on a single workstation. The first step the program takes is to load parameters from a configuration file or the runtime system. Once all the parameters have been found, the program creates a two-dimensional matrix to hold the incoming signal data. The columns of the matrix correspond to data from the different array nodes, and each row is a time slice of the samples taken, as shown in Figure 7.3.2. The program then proceeds with the algorithm presented in the previous algorithm section; it calculates the windowing functions and multiplies the columns of data by the appropriate nodes' factors. Next, the program takes the Fast Fourier Transform of the data column by column, which was a subroutine excerpted from [MORG94]. The algorithm then enters a loop which executes once for each steering direction. The steering directions are dictated by the values the user entered for the search angle extremes and the incremental angle to search, such as between -90° and 90° in increments of 2°. For each iteration of the loop, the program multiplies each column of the transformed data by a node-dependent steering factor value, sums all the columns of the data, and inverse transforms the resulting summed column. Figure 7.3.3 shows the flowchart for the sequential program. It is this program which is manipulated to create the baseline and the various parallel programs.

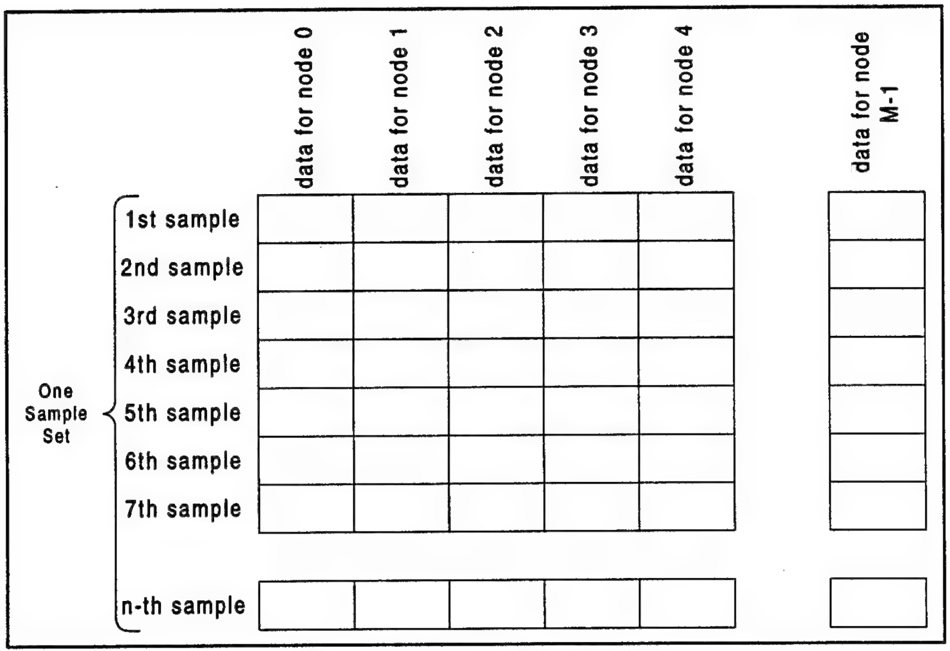


Figure 7.3.2 : Input Data Matrix

The data matrix is set up so that each node (0 through M-1) has a column.

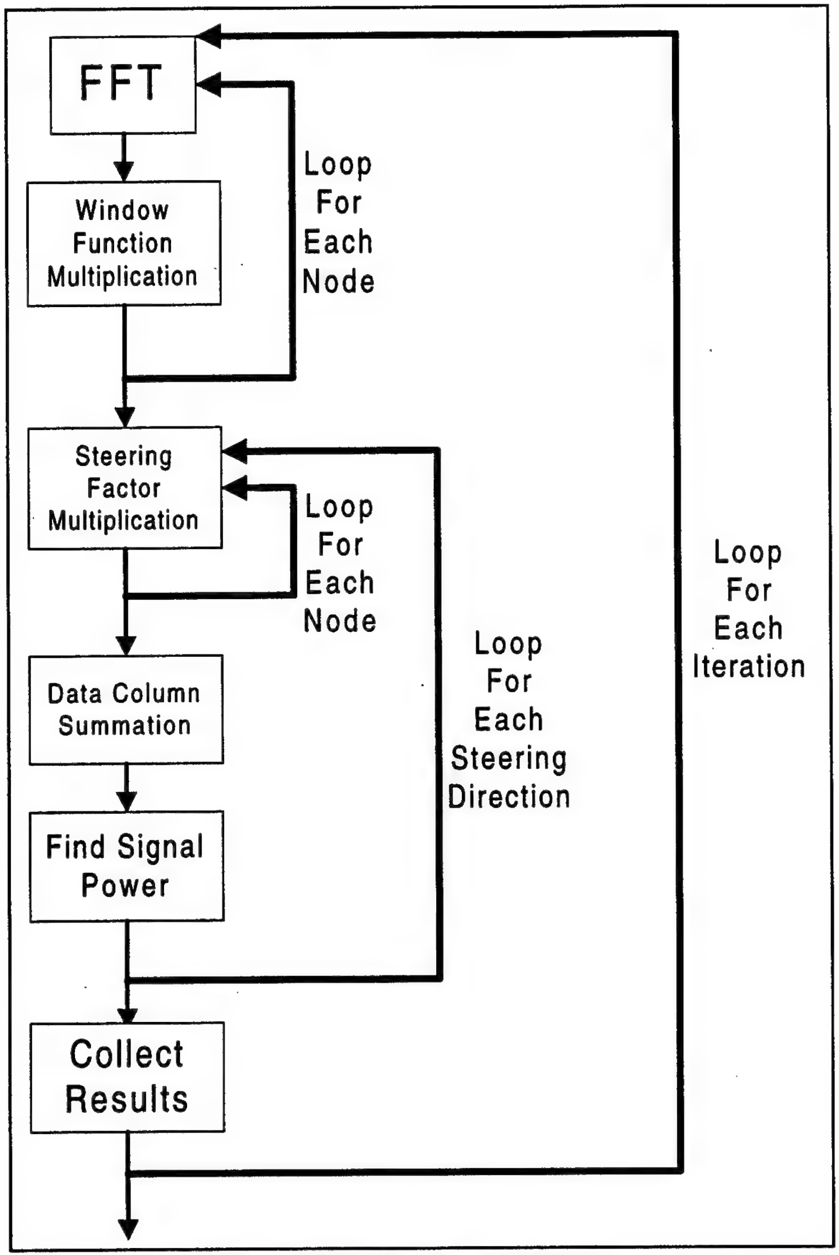


Figure 7.3.3 : Flowchart for SEQFFT

Though this flow chart was shown in Figure 5.5.2, it is presented again here for reference.

Timing considerations for the baseline and parallel programs are critical. First, accessing the configuration file is excluded from the timing. Also excluded is the time the computer takes to create the signal data because it is assumed that the signal data in the sonar array will be placed into memory from the audio transceiver without computational interference from the processor. The graphical presentation of the results is, of course, also not timed. Second, the fact that a single iteration of the algorithm may take on the order of a second causes a timing concern. It is not desired to have timings taken for only a few seconds because any number of factors in the simulating processor may cause timing errors as large as the total execution time. In all of the algorithms implemented, many iterations (e.g. 100) were run to get larger timing numbers to absorb variations between iterations. The times were then divided by the number of iterations in order to get an average iteration time. It is important to note that not everything in the algorithm presented above is included within the loop for the several iterations. Again, the configuration file access and the input signal computation are left out so that only the window factor, FFT, and the steering direction loop are included.

7.3.2 Baseline Specification

It is important to specify the baseline used for the parallel algorithms in the frequency-domain so that valid comparisons can be made. The method it follows is that of a collection of dumb nodes whose only job is to collect data with an audio transceiver and send it forward. This communication is implemented by the method previously discussed in which the nodes send one link at a time in order to make a rudimentary simulation of the underlying unidirectional linear array architecture. The most upstream node sends its column of data down to the next node downstream. This node appends its data column to that of the upstream node and sends both columns downstream. The next column downstream again appends and sends until the front-end node has the entire matrix. The front-end processor of the baseline then performs all the calculations on the data as dictated by the sequential algorithm.

To implement this, a thread is made for each node in the array plus another separate thread for the front-end processor. The array node threads are given a standard function to run and are passed the array node index so they can distinguish each other. The threads are spread across processors in the simulation testbed. Another thread running another function is created on one of the testbed processors to represent the front-end processor. Once all the communication to the front-end is completed, the algorithm proceeds just like the sequential algorithm. The topology and the simulation model for the baseline are illustrated in Figure 7.3.4.

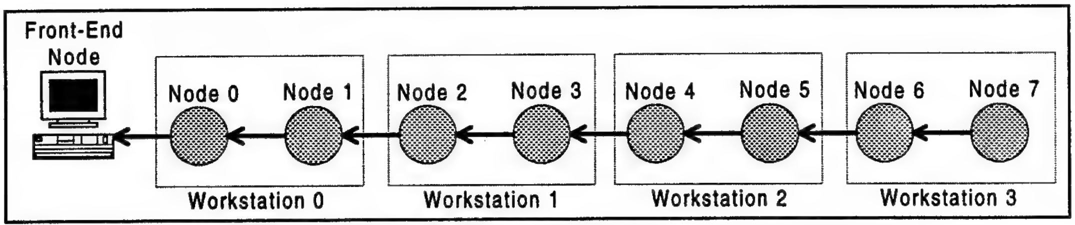


Figure 7.3.4 : Implementation Model for the Frequency-Domain Baseline

The model for implementing the algorithms is that several nodes (threads) are created across a distributed workstation testbed. One thread is created for the front end.

The baseline introduces an important concept in the simulation of the sonar array. Rather than having each node use an MPI function call to send data directly to the front-end processor, the nodes are required to simulate a unidirectional linear array. Each node sends its data only to the node downstream from it, which then receives the data packet and places its own data onto the end of the packet to again send downstream. This process continues until the front-end processor receives a packet containing all data columns. This extra complexity in the communication results in overhead which is to reflect the latency in the underlying network architecture. Though not a fair representation of the physical time the network will take, this method does provide for a useful baseline for comparison because the parallel algorithms use the same method.

There are a few points to make about how the baseline performs. As more nodes (threads) are added to the array, the execution time of the baseline increases. Furthermore, as more workstations are added with the same number of node threads, the execution time also slows. This is due to the extra inter-workstation communication and fewer inter-workstation communications. Also of interest is how the baseline performance improves by using SCI instead of Ethernet. This is due to the better throughput and lower latency of the SCI interconnect over Ethernet. The use of HCS threads over Solaris threads also improves performance due to the fact that the HCS threads remove much of the overhead of the Solaris threads, such as kernel context switches. All the upcoming parallel algorithms will be compared to the baseline which matches the interconnect and the thread library used in the parallel algorithm. For example, the timings for the baseline FFT over Ethernet using HCS threads will be used in meaningful comparisons with the parallel ring algorithm over Ethernet using HCS threads but not with the parallel ring over SCI or using Solaris threads. In this way, valid comparisons can be made without needing to strictly quantify the time contributions from each factor.

7.3.3 Parallel Unidirectional FFT Beamformer

The first of the parallel algorithms implemented was the parallel unidirectional FFT, or PUF. As discussed in the algorithm decomposition section, this algorithm has little possible parallelism due to the unidirectional nature of the underlying network. The most downstream node must always do considerably more work than the other nodes, thus the speedup is not expected to be good. The speedups versus the baseline for the two versions, PUFv1 and PUFv2, are presented here.

The first version of the parallel unidirectional FFT is PUFv1, which uses the data-parallel mechanism in which the nodes transform their values and send them downstream. The first node then finishes the algorithm. The threads send their data down on a dynamically growing train to the first node link by link using the abstract communication calls, just as in the baseline. There is no separately created front-end thread in this algorithm. Instead, all threads execute the same function and can be distinguished only by the node index number it was passed during initialization. The thread which realizes it is node 0 will receive the data from upstream and then carry out the loop for all the different steering directions.

The communication pattern for PUFv1 is not the same as the communication pattern for the baseline, but they are comparable. In the baseline program, if there are 32 samples taken at each node, then each column in the communicated matrix has 32 rows. The parallel unidirectional program differs because after the FFT, only 16 samples out of a 32-point FFT contain useful data. The other 16 values are simply mirror images of the first 16 samples, so the columns of the matrix only contain rows for the useful 16 values. Though PUFv1 communicates fewer samples per column, each row in the column is now a complex number because the transform outputs complex numbers. In the implementations for this project, complex numbers are comprised of two single-precision floating point numbers. Therefore, the communications for the baseline and PUFv1 send the same number of floating point numbers, and any comparisons made between them are valid.

There is a granularity knob built into the PUFv1 program which controls the size of the packets in the downstream communication. Specifically, the granularity parameter controls how many columns of the data matrix are sent at a time. If the knob is set to 4, the nodes will group themselves into 4 nodes per group. Each group sends its data down to the most downstream node in the group which collects the columns into a packet of 4 columns. This 4-column packet is then sent down intact through all the nodes until it reaches the first node of the array. This process is shown schematically in Figure 7.3.5. Decreasing the granularity knob means fewer columns are collected into each packet, and increasing the granularity knob results in more columns collecting into larger packets. On the fine-grained extreme, when the granularity knob is set to 1, all nodes send down their column immediately, then wait for a single column from their upstream neighbors. They forward this column down and wait for another from upstream, until all columns have trickled down to the first node. On the coarse-grained extreme, when the knob is set to equal the number of nodes in the array, the communication behaves exactly like the communication of the optimized growing train in the baseline program.

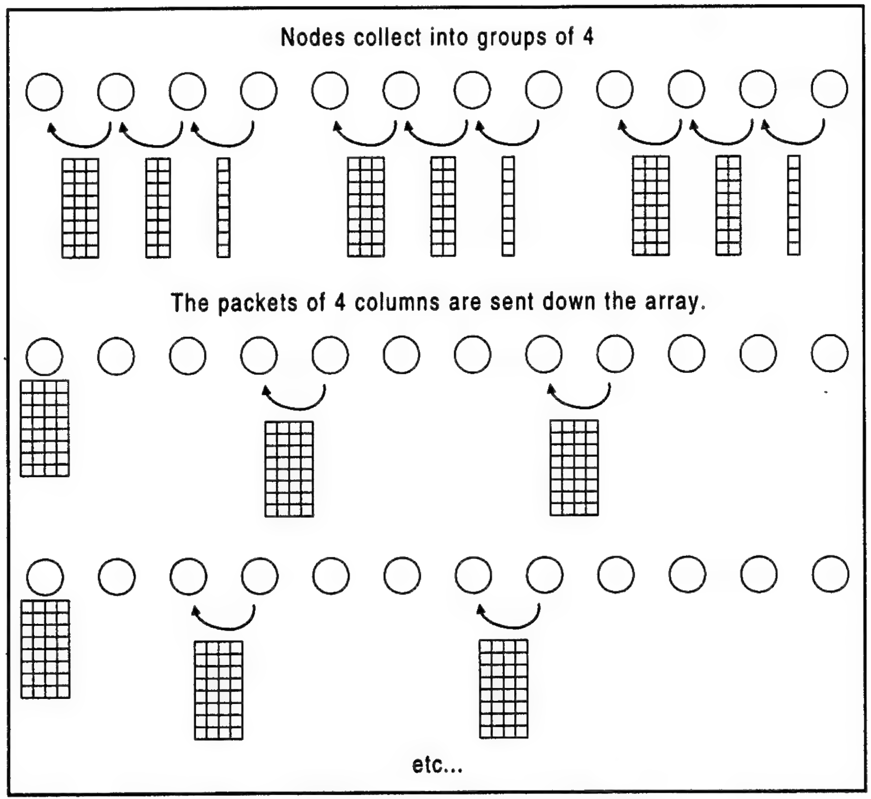

Nodes collect into groups of 4
The packets of 4 columns are sent down the array.
etc...

Figure 7.3.5 : Granularity in Nodes per Packet

This figure illustrates the procedure when the granularity knob for nodes per packet is set to 4. Columns are collected in groups of 4 before trickling down the array.

There is an advantage to running the algorithm with a coarse granularity due to the overhead involved in handling the finer-grained communication. In the finest-grained case, each node must initiate a send to transmit its column, then initiate a receive to get the column from the node upstream. A send must then be initiated to transmit this column downstream. The pattern of initiating sends and receives until all columns are down to the first node can cause significant overhead. In the medium-grained case, there is overhead in that the nodes must determine who is in their groups. Once the groups are recognized and the most downstream node of the group collects the packet, there is still the matter of getting the packet to trickle down. This requires the attention of all nodes to forward the packet downstream. The algorithm set to the coarsest-grain possible is the most efficient algorithm. All that is needed is for a node to wait for a packet from upstream, append its data to the packet, and send the packet on its way downstream. For each node in the coarse-grain approach, only one receive and one send must be initiated.

Even with the granularity knob set to the coarsest possible grain, the algorithm's performance is not spectacular. Figure 7.3.6 and Figure 7.3.7 show the speedup of the Ethernet version of PUFv1 over the Ethernet version of the baseline and the speedup of the SCI version of PUFv1 over the SCI version of the baseline, all with Solaris threads. The SCI version generally gets speedups in the area of 1.2, but the Ethernet version does not get any speedup—only in the range 0.95 to 1.0. In fact, due to the inclusion of code to implement the granularity knob, PUFv1 performs slightly worse than the baseline. Also, the communication time dominates the Ethernet version so that the parallelization of the FFT and window factor multiplication has hardly any effect. The SCI comparison shows this first version of the parallel unidirectional is still a better performer than the baseline, though not by much. Considering that the true sonar array will have a network throughput and latency closer to that of the Ethernet testbed than that of the SCI testbed, it is clear that PUFv1 will not deliver outstanding performance, although it will suffice over the baseline.

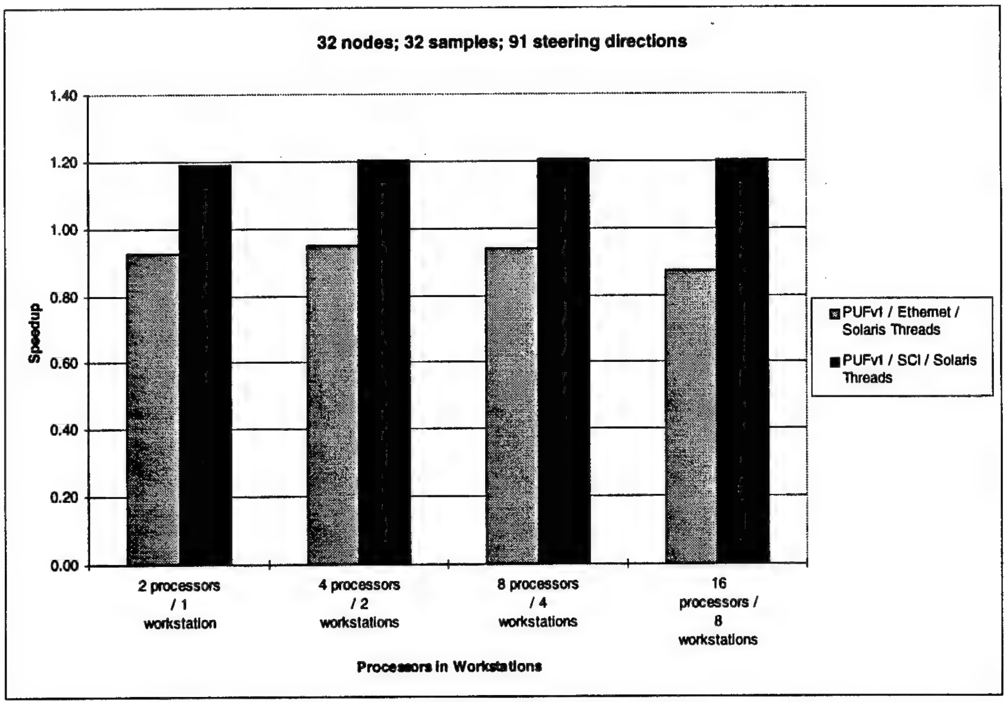


Figure 7.3.6 : Speedup - PUFv1 vs. Baseline - small problem size

Speedup of the first version of the parallel unidirectional FFT beamformer (PUFv1) over the baseline with 32 nodes, 32 samples per FFT, and 91 steering directions.

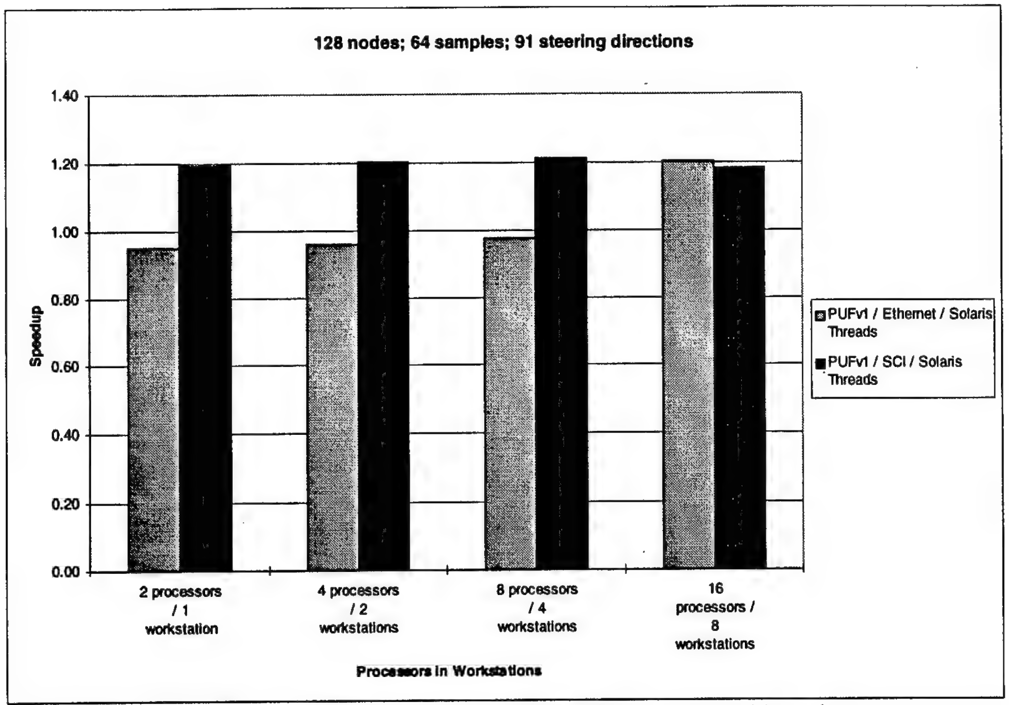


Figure 7.3.7 : Speedup - PUFv1 vs. Baseline - large problem size

Speedup of the first version of the parallel unidirectional FFT beamformer (PUFv1) over the baseline with 128 nodes, 64 samples per FFT, and 91 steering directions.

The next program is the second version of PUF, PUFv2, which improves of PUFv1 by changing the communication pattern. As previously discussed, the summation occurs during the communication instead of within the front node, thus increasing the amount of possible parallelism; however, the communication/summation mechanism must be executed as many times as there are steering directions. The timing numbers show that PUFv2 provides a considerable improvement over PUFv1. Although performance on one workstation is not good, the performance improves as more processors are added, as shown in Figure 7.3.8, Figure 7.3.9, and Figure 7.3.10. For a single workstation running this program, speedups over the baseline were in the range 0.28 to 0.6; however, the positive trend can be seen by noting that speedups over the baseline for eight workstations range from 1.4 to 7. This trend illustrates the scalability of the algorithm, which means that as more processors are added, the algorithm will continue to improve performance; however, it is far from a linear speedup. Though the performance will continue to improve as processors are added, the factor improvement will not be as large as the factor increase in the number of processors.

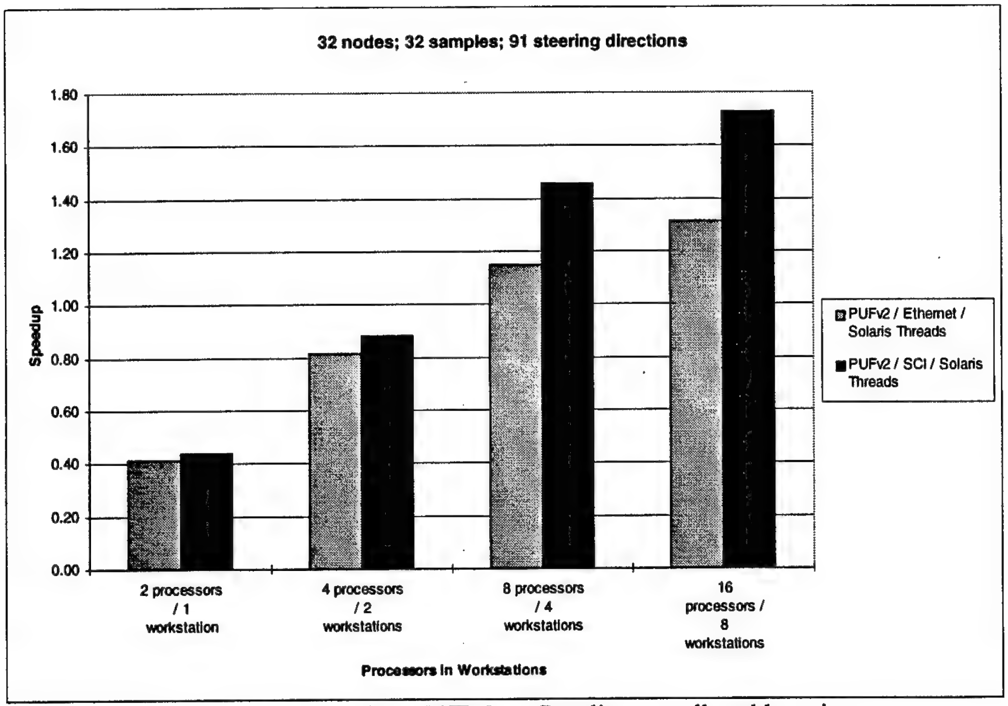


Figure 7.3.8 : Speedup - PUFv2 vs. Baseline - small problem size

Speedup of the second version of the parallel unidirectional FFT beamformer (PUFv2) over the baseline with 32 nodes, 32 samples per FFT, and 91 steering directions.

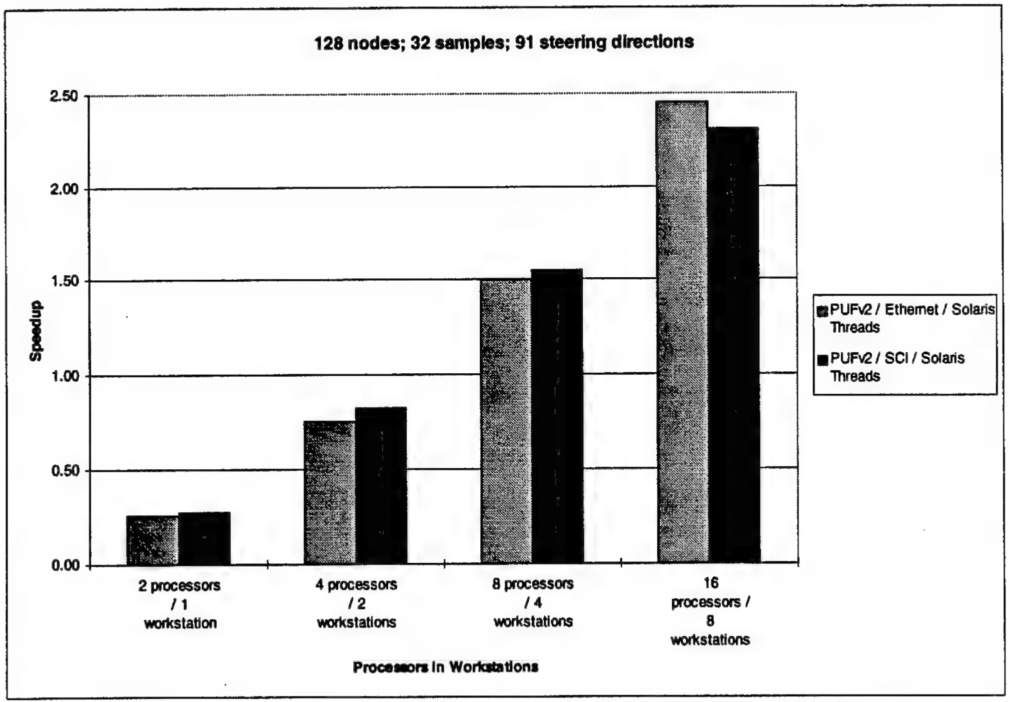


Figure 7.3.9 : Speedup - PUFv2 vs. Baseline - medium problem size

Speedup of the second version of the parallel unidirectional FFT beamformer (PUFv2) over the baseline with 128 nodes, 32 samples per FFT, and 91 steering directions.

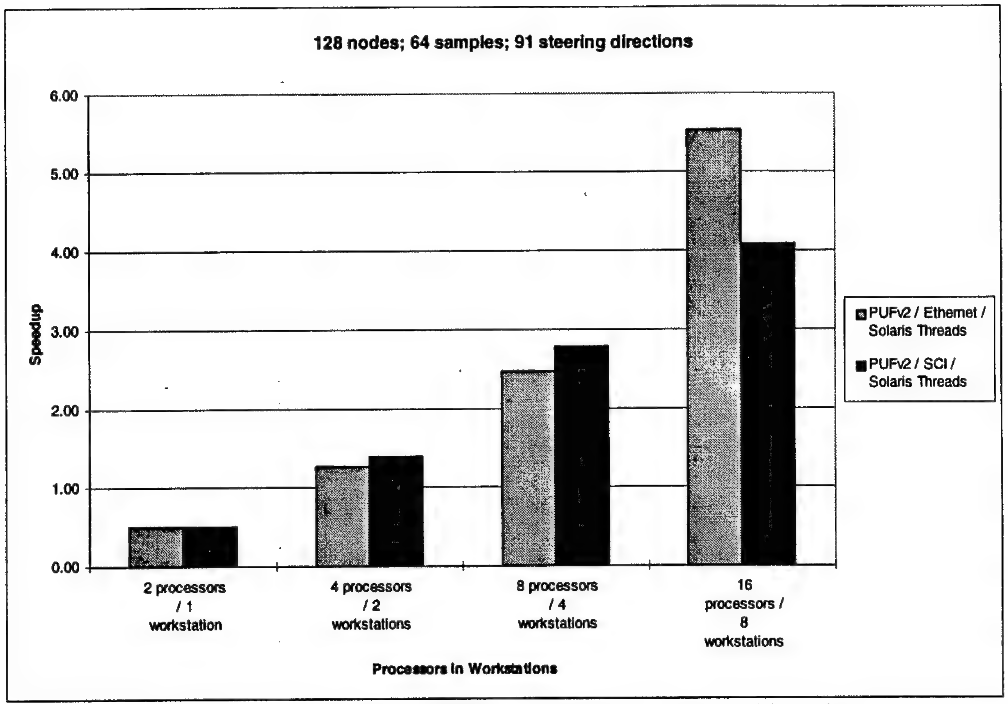


Figure 7.3.10 : Speedup - PUFv2 vs. Baseline - large problem size

Speedup of the second version of the parallel unidirectional FFT beamformer (PUFv2) over the baseline with 128 nodes, 64 samples per FFT, and 91 steering directions.

Also of interest is that the SCI version never outperforms the SCI baseline as much as the Ethernet version outperforms the Ethernet baseline. This illustrates how the communication costs were the major contributor to the Ethernet baseline. Since the communication in PUFv2 is much less than the communication in the baseline, the performance improvement is more noticeable over the slower interconnect, which is Ethernet.

7.3.4 Parallel Ring and Parallel Bidirectional FFT Beamformers

The next programs, the parallel ring FFT beamformer (PRF) and the parallel bidirectional FFT beamformer (PBF), were developed with underlying architectures other than a unidirectional linear array, which allows for more efficient programs since they can pipeline the iterations with a float front-end.

In these FFT programs, the selection order for which node will be the front-end for which iterations can be somewhat specified by the user via a granularity knob built into the program. The basic order is the order of the nodes in the array, where for the first iteration, node 0 is the specified front-end, on the next iteration is node 1, then node 2, and so on. This is the case when the granularity knob is set to 1. Increasing the granularity makes the selection algorithm skip a few nodes before assigning another front-end. For example, with the granularity set to 4, then every 4th node in the array will be specified the front-end as the iterations complete. This knob is built in because it is desirable in the simulation to represent the array conditions as much as possible. The granularity allows the user to set the front-end nodes according to the processors on the testbed. For example, if each processor running the simulation is allocated 8 threads, then the granularity can be set to 8 so that floating front-end nodes can each run on a processor. It would be unfair of the simulation to force 8 successive front-ends onto the same processor to be time-shared while the other processors remain relatively idle because in the final array, each node will have its own processor.

The first program timed is the parallel ring FFT. The program takes the algorithm from PUFv1 and implements the floating front-end with the pipelining of front-end work and other iteration initiations. Each node determines who is the front-end by using the specified formula and the value of the granularity knob. Communication with the dynamically growing train then proceeds down the ring link by link to emulate the ring architecture. The PRF program also supports the granularity knob for the packet size, as did PUFv1. The nodes in the ring group into whatever size group the knob dictates, collect their data into a packet of that many columns, and send the entire packet down as one entity. As in the PUFv1 case, the coarse-grained approach, where the knob was set to the number of array nodes so that the communication would become a growing train, proved to be much more efficient than the finer-grained approaches, and so the coarse-grained timing numbers are those taken into consideration.

The performance of the parallel ring FFT is considerably better than any of the previous algorithms. In fact, PRF scales very well and has a speedup curve approaching linear; that is, for 16 processors, the speedup gets up into the teens, as opposed to the speedups for the parallel unidirectional FFT which stayed in single digits. The speedup is charted for PRF against the baseline in Figure 7.3.11 and Figure 7.3.12.

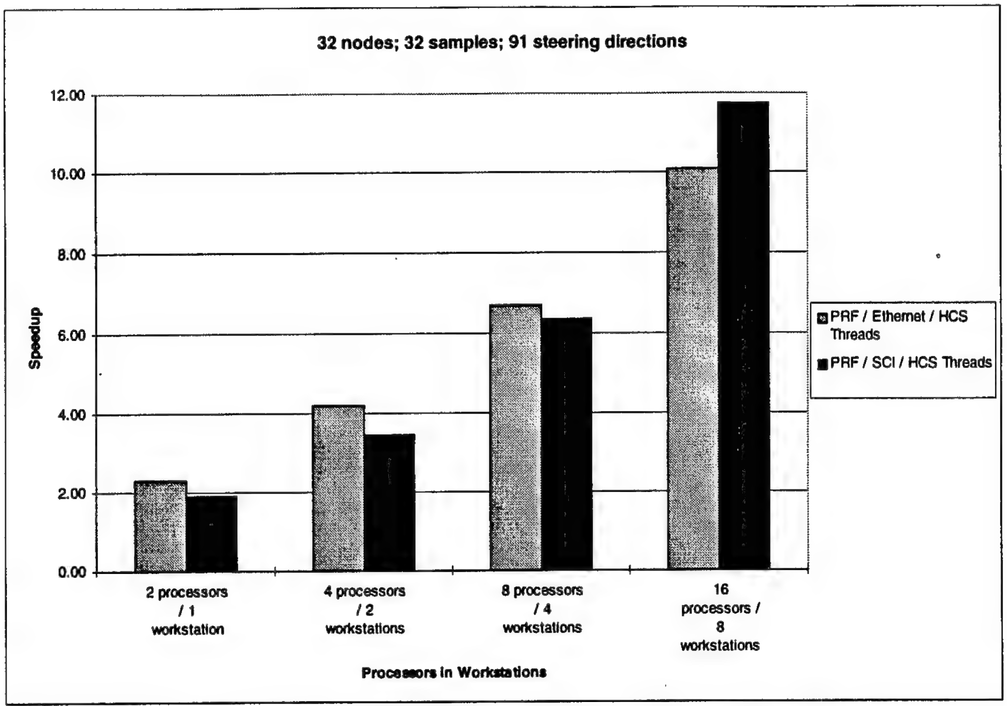


Figure 7.3.11 : Speedup - PRF vs. Baseline - small problem size

Speedup of the parallel ring FFT beamformer (PRF) over the baseline with 32 nodes, 32 samples per FFT, and 91 steering directions.

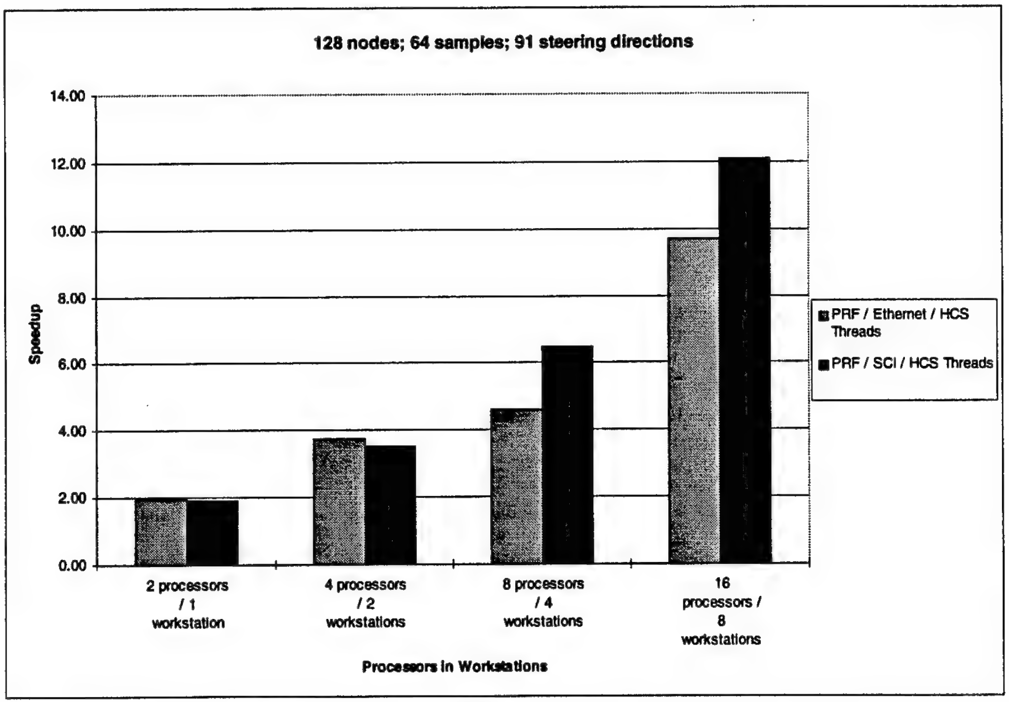


Figure 7.3.12 : Speedup - PRF vs. Baseline - large problem size

Speedup of the parallel ring FFT beamformer (PRF) over the baseline with 128 nodes, 64 samples per FFT, and 91 steering directions.

The last program in the frequency domain is the parallel bidirectional FFT (PBF). The timings show that PBF also performs very well against the baseline, just as did PRF. The speedups for PBF versus the baseline are given in Figure 7.3.13 and Figure 7.3.14.

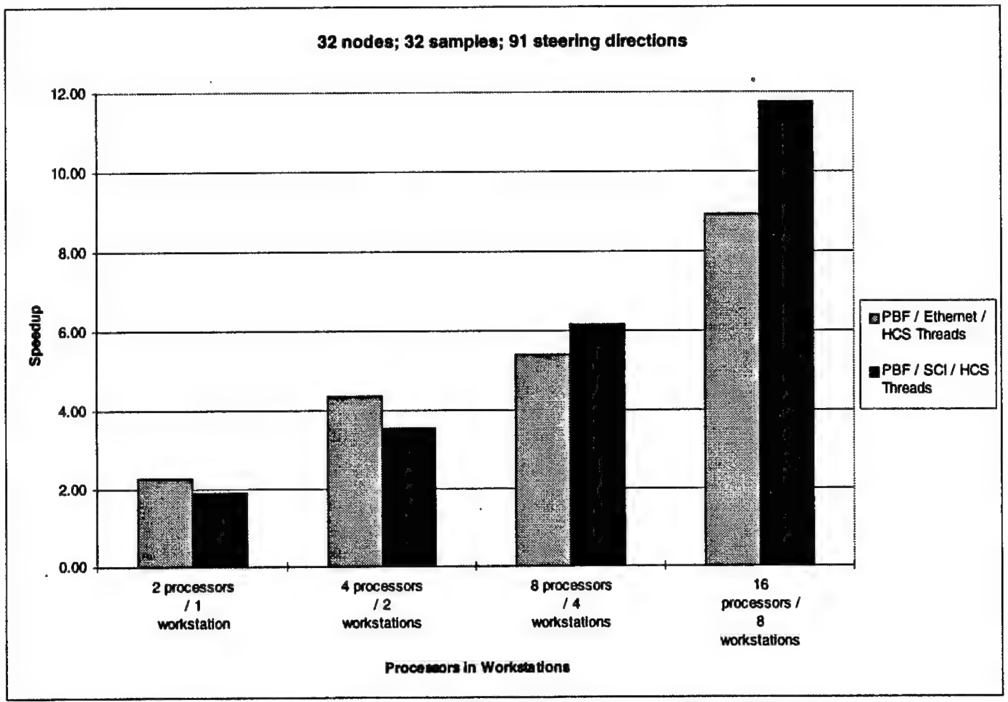


Figure 7.3.13 : Speedup - PBF vs. Baseline - small problem size

Speedup of the parallel bidirectional FFT beamformer (PBF) over the baseline with 32 nodes, 32 samples per FFT, and 91 steering directions.

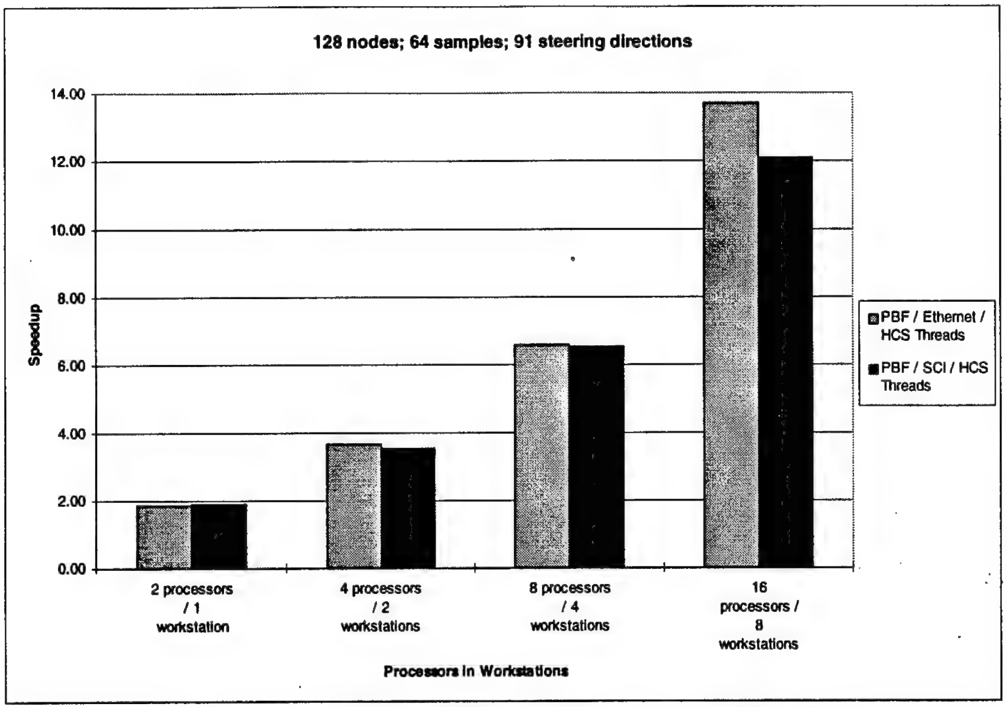


Figure 7.3.14 : Speedup - PBF vs. Baseline - large problem size

Speedup of the parallel bidirectional FFT beamformer (PBF) over the baseline with 128 nodes, 64 samples per FFT, and 91 steering directions.

All these programs, though run on testbeds that are not necessarily representative of the final sonar array, provide valuable insights into the methods of decomposition and of simulation. In the following section, the general methods of the parallel algorithms are compared with each other. Also, the best case times for the various algorithms are given and compared with the times for the other algorithms running the same parameters. Also given is a discussion of the overhead created by the various methods, including the overhead due to the simulating threads and that due to communication.

7.3.5 Parallel Program Comparison

The previous discussions emphasized the speedups of the various algorithms over the baseline. It was seen that the two versions of the parallel unidirectional FFT provided a slightly better performance than the baseline. The parallel ring algorithm and the parallel bidirectional algorithm provide considerable improvement over the baseline. Rather than observing their speedups versus the baseline, direct comparisons are made in the next section between the algorithms by making general observations and by looking at the timing numbers. By doing this, it can be determined why an algorithm is faster than another, how the slow algorithms can be fixed, and which algorithms are worth examining in more detail. The speedups of the parallel versions of all the programs for a typical scenario are placed next to each other in Figure 7.3.15, Figure 7.3.16, Figure 7.3.17, and Figure 7.3.18. Algorithms deemed suitable will need to be simulated with the upcoming BONeS/MPI interface so that exact timings may be made for the true hardware.

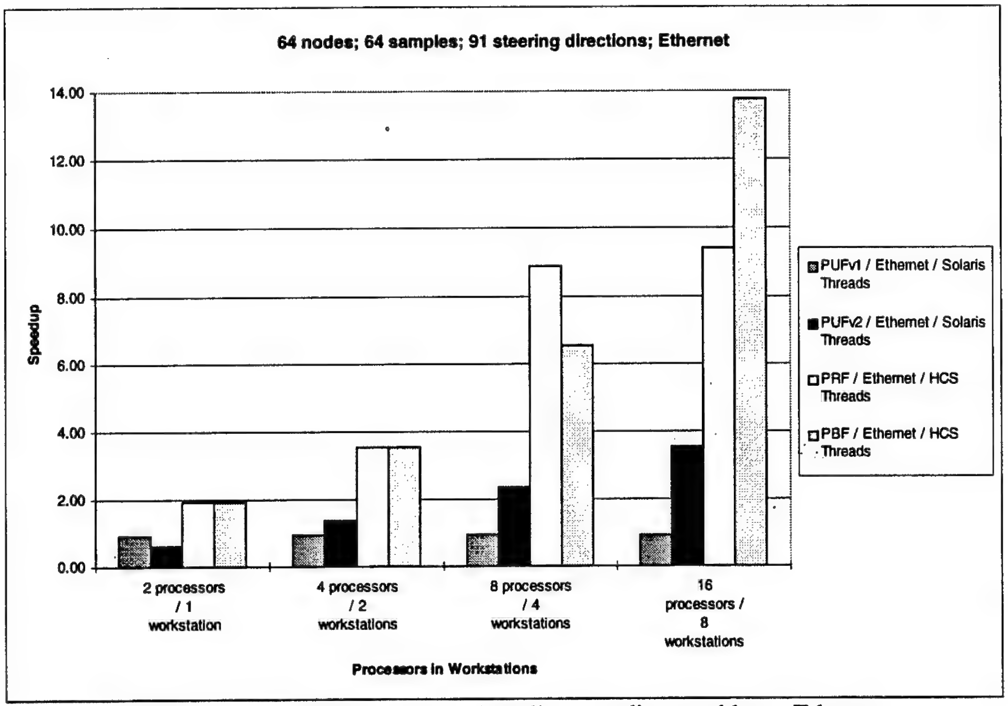


Figure 7.3.15 : Speedup - All vs. Baseline - medium problem - Ethernet

Ethernet speedup of all the parallel programs over the Ethernet baseline with 64 nodes, 64 samples per FFT, and 91 steering directions.

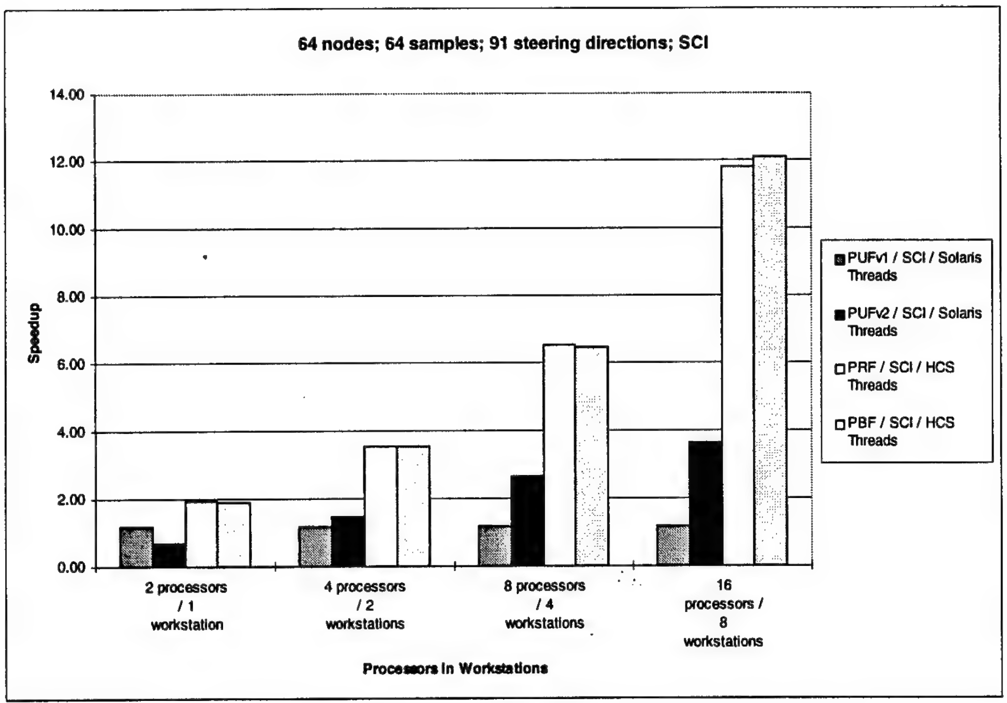


Figure 7.3.16 : Speedup - All vs. Baseline - medium problem - SCI

SCI speedup of all the parallel programs over the SCI baseline with 64 nodes, 64 samples per FFT, and 91 steering directions.

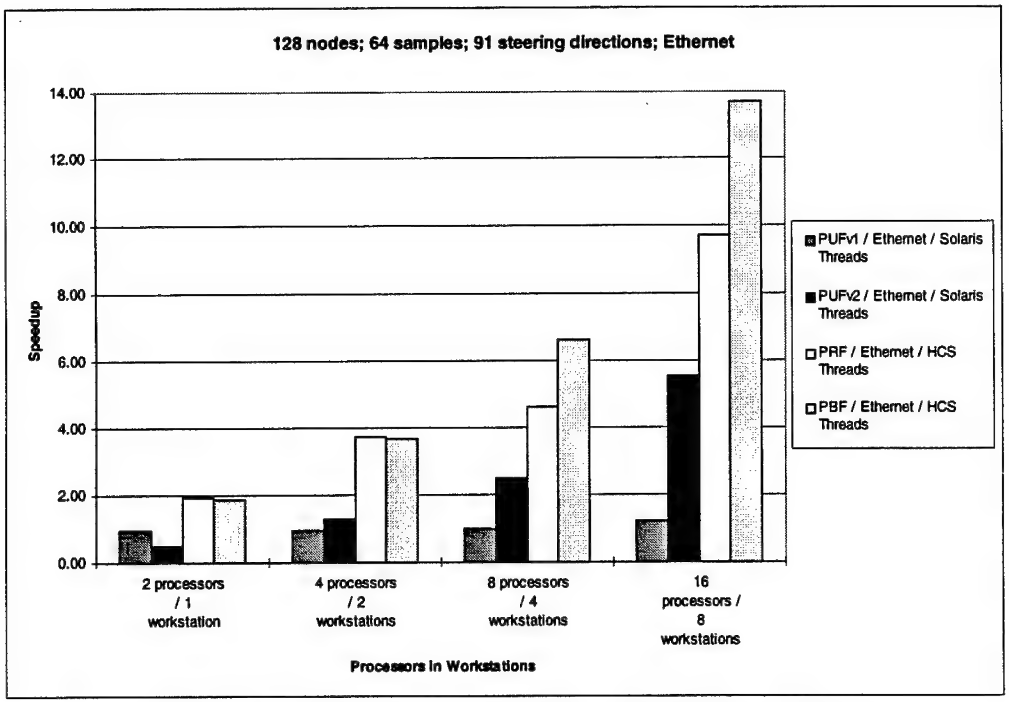


Figure 7.3.17 : Speedup - All vs. Baseline - large problem - Ethernet

Ethernet speedup of all the parallel programs over the Ethernet baseline with 128 nodes, 64 samples per FFT, and 91 steering directions.

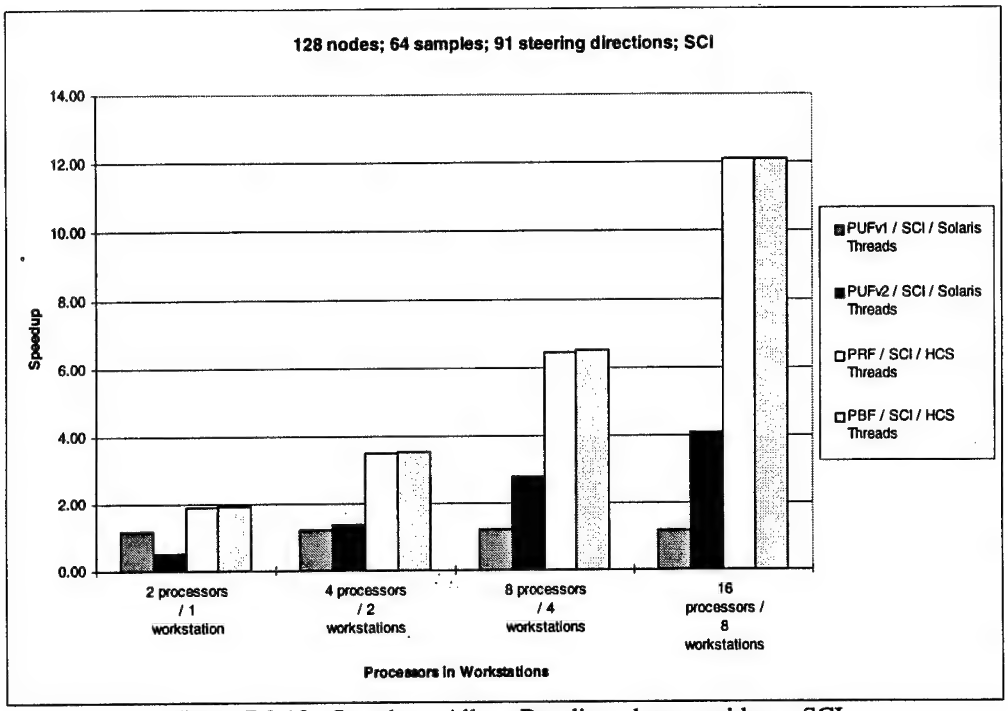


Figure 7.3.18 : Speedup - All vs. Baseline - large problem - SCI

SCI speedup of all the parallel programs over the SCI baseline with 128 nodes, 64 samples per FFT, and 91 steering directions.

The first parallel algorithm implemented was PUFv1, which did not provide particularly good speedup. Due to the fact that PUFv2 does its summation before the communication, PUFv2 outperforms PUFv1 in cases where the problem size is large enough to offset the cost of putting the steering direction loop into the hands of the several nodes. Furthermore, the use of more processors helps PUFv2 much more than PUFv1 because of the increased parallelism in PUFv2. The speedup of PUFv2 over PUFv1 can get up into the range of 3.5 to 4.5. The ring (PRF) and bidirectional (PBF) programs outperform PUFv1 by factors as much as 9 to 10 simply due to the fact that these algorithms are not constrained by the requirement that the first node do the bulk of all work.

The next program created was PUFv2, which introduced a more efficient unidirectional algorithm, and so it compares better with the ring and bidirectional algorithms that did PUFv1. However, the ring and bidirectional algorithms still provide considerable speedup above PUFv2. PUFv2 does outperform the ring and bidirectional algorithms as far as scalability is concerned. The performance of the ring and bidirectional algorithms does not improve as fast as PUFv2 with the addition of more processors. This is due to the fact that PUFv2 uses a more efficient communication algorithm, and the ring and bidirectional programs use the growing freight train, which will take more time to communicate as more intra-workstation communications are replaced by inter-workstation communications.

The comparison between the ring and bidirectional programs is not as decisive as comparisons with the PUF versions. Although the bidirectional shows performance improvements over the ring in some cases, the ring and bidirectional array algorithms do not clearly have a performance winner. This is because only the Ethernet version shows the performance increase in the bidirectional; the SCI timings show no such improvement. Obviously, in a sonar array architecture which is not built yet, the true performance may be more like the Ethernet version or more like the SCI version. An attempt might be made at this point to figure out analytically which algorithm would perform better. For example, the ring PRF uses a one-degree interconnect for the underlying architecture, meaning each node only has one network path through it, one input link and one output link. The parallel bidirectional FFT algorithm uses a two-degree network, which means there are two input links and two output links. When communication is heavy, such as in the case of the large problem size, the software simulation for the bidirectional program has more overhead to handle the two possible incoming data directions. Also, the software simulation in the case of these two algorithms cannot be declared as a perfect simulation of the true architecture. For example, even though the simulation will always have more overhead when it handles communication in two directions, the hardware may not have the same overhead. If the links used in the bidirectional array have the same bandwidth as the links in the ring and the link must be shared between the two directions, then the bidirectional nature of the links will cause a slow-down. However, if for each direction in the bidirectional array, the links have as much bandwidth as the single-direction link in the ring, then there is no reason to believe why the bidirectional algorithm will exhibit a performance lag behind the ring algorithm. This is due to the fact that there is more work involved in determining which direction to send data, and there is more work in the framework for the abstract communication to handle the two directions.

Between the ring algorithm and the bidirectional algorithm, there is no clear choice for the best performing algorithm. In these two algorithms, it will be imperative to simulate the code more precisely by using the BONeS/MPI interface. Since no choice can be made from the software simulation, it will be necessary to have the program run over the BONeS simulation of the exact network to be studied. In this way, it can be determined which algorithm will work best for a specific network and exactly how much better it is. Further research in the upcoming year into integrating an MPI program with BONeS will be needed to resolve this issue.

7.3.6 Parallel Speedup Over Sequential

The previous discussion concentrated on the results and speedups obtained versus the baseline. This baseline was formulated to not only emulate the current method of rapidly-deployable sonar arrays but also to create a communication and algorithm model which can be easily compared to the parallel programs. Comparison with the baseline gives a measure of how the parallel programs will perform on an array. However, there is another performance metric for parallel programs which does not involve a baseline rooted in a sequential unidirectional algorithm. Instead, it is beneficial to compare the parallel programs with a purely sequential program. Examining speedup times over a purely sequential program is the method most often used in the computer engineering realm of algorithm development. Although it is not as important for the goals of the sonar array project as the speedup over the baseline, the speedup over the purely sequential program is still of interest to algorithm developers. The speedup over the purely sequential program may also be of interest in off-line beamforming of previously collected data.

Depending on the thread library and interconnection used on the testbed, the baseline performed at between 50% to 85% of the purely sequential performance. This section will relate the execution times and speedups of the parallel programs over this sequential program.

For the small problem size of 32 array nodes working with 32-point FFTs, the execution times are shown in Figure 7.3.19. All execution times are given as average-per-iteration values because many iterations were run to average out any abnormalities in any one iteration. On the horizontal axis, each group shows the values for a particular program, and bars within a group show how many processors were used to obtain the value in that bar. The speedups for these same programs over the purely sequential is then given in Figure 7.3.20.

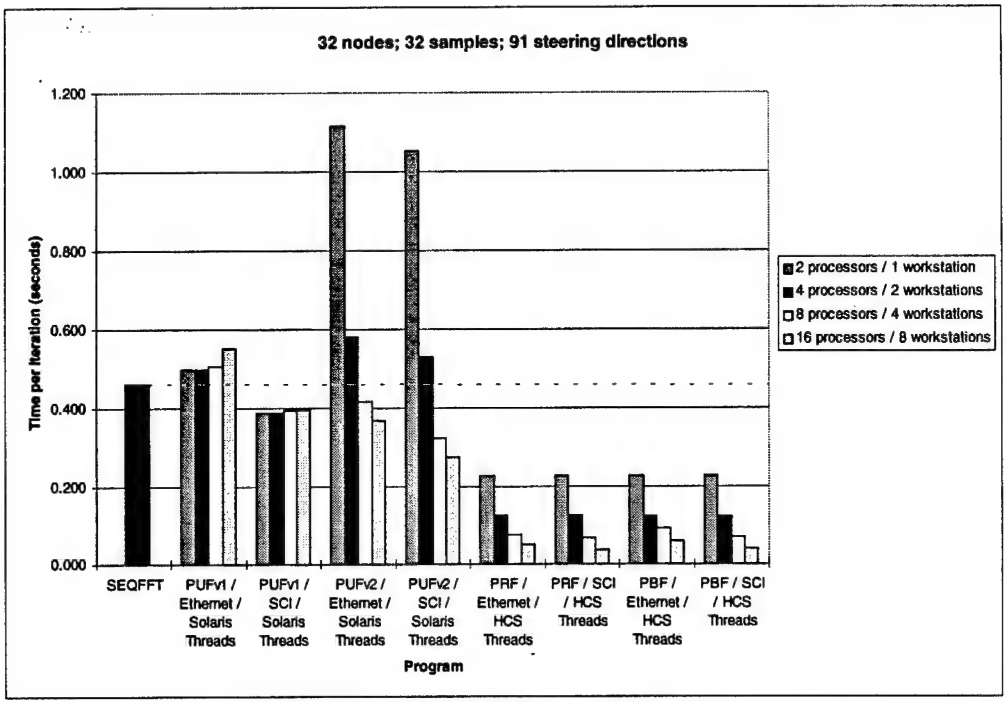


Figure 7.3.19 : Execution Times - small problem size

This graph shows execution times of all parallel programs and the purely sequential program (SEQFFT). The horizontal axis gives each program. Bars within the groups give the number of processors in the number of dual-CPU workstations, as shown in the legend.

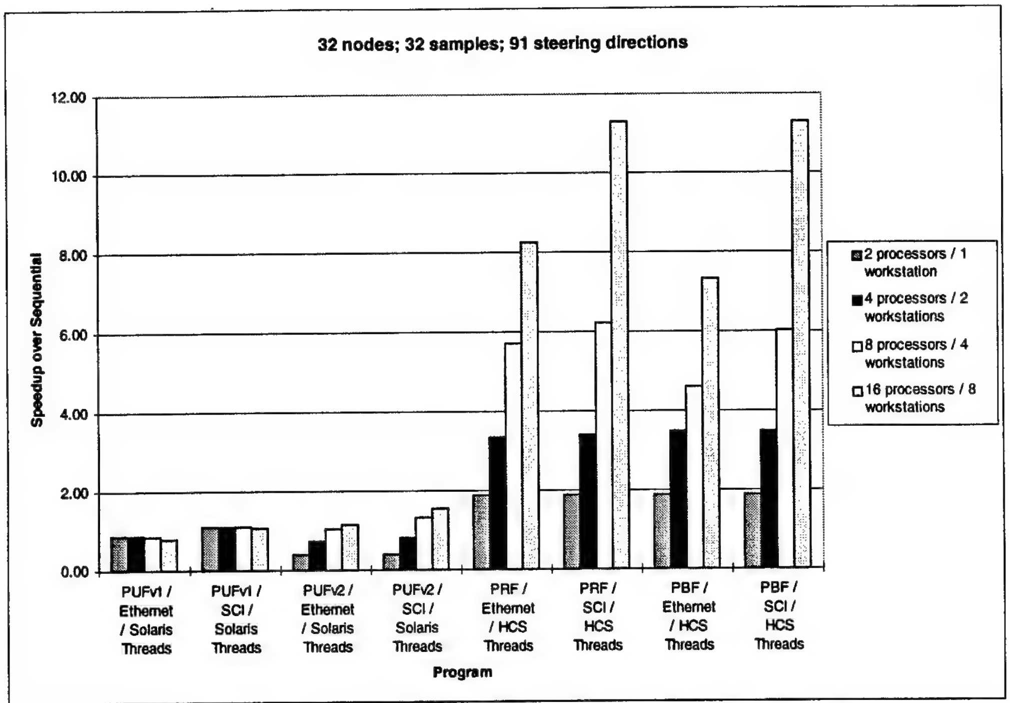


Figure 7.3.20 : Speedup vs. Pure Sequential - small problem size

Speedups of all parallel programs versus the purely sequential program (SEQFFT) for 32 nodes, 32 samples per FFT, and 91 steering directions.

The speedup graph shows several important concepts in the parallel programs. First, the speedups for the parallel ring FFT beamformer (PRF) and the parallel bidirectional FFT beamformer (PBF) provide considerably better performance than any of the other programs. This shows the exceptional advantage in having a fully-connected network as opposed to the unidirectional network of the PUF programs. Second, the parallel programs almost always perform better over SCI than they do over Ethernet, the exception being the case of 1 workstation where neither SCI nor Ethernet is used. The most glaring improvements provided by SCI show up in the 8 workstation situation because it includes the most communication over the testbed. Third, all the parallel programs except PUFv1 provide good scalability. This includes PUFv2, because even though it does not perform as well as the programs based on the fully-connected network architecture, it does increase its performance as processors are added. Another way of stating this is that its efficiency does not decrease nearly as much as the efficiency of PUFv1. This efficiency concept is illustrated in Figure 7.3.21.

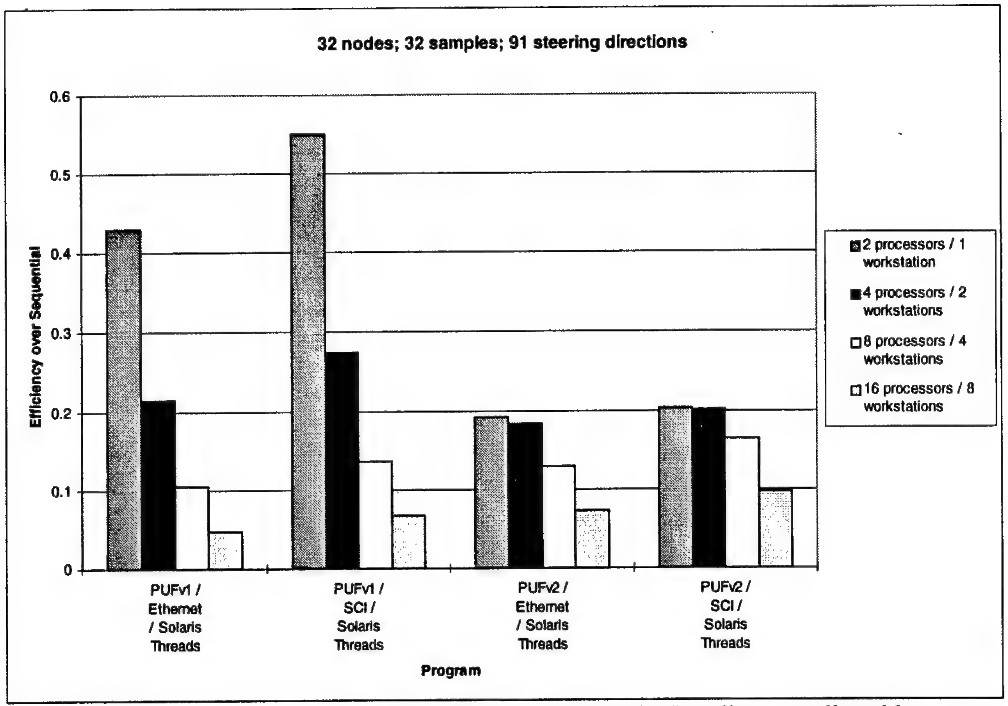


Figure 7.3.21 : Efficiency - PUFv1 and PUFv2 over the Baseline - small problem

This graph shows efficiency, which is defined as speedup divided by number of processors (number of processors equals the ideal speedup).

Executions of the large problem size present almost the same results, thereby confirming the best algorithms remain good no matter the problem size. The execution times for the programs running the large problem size of 132 array nodes and 64 samples per FFT are shown in Figure 7.3.22. The speedups of these algorithms versus the purely sequential FFT beamformer are shown in Figure 7.3.23.

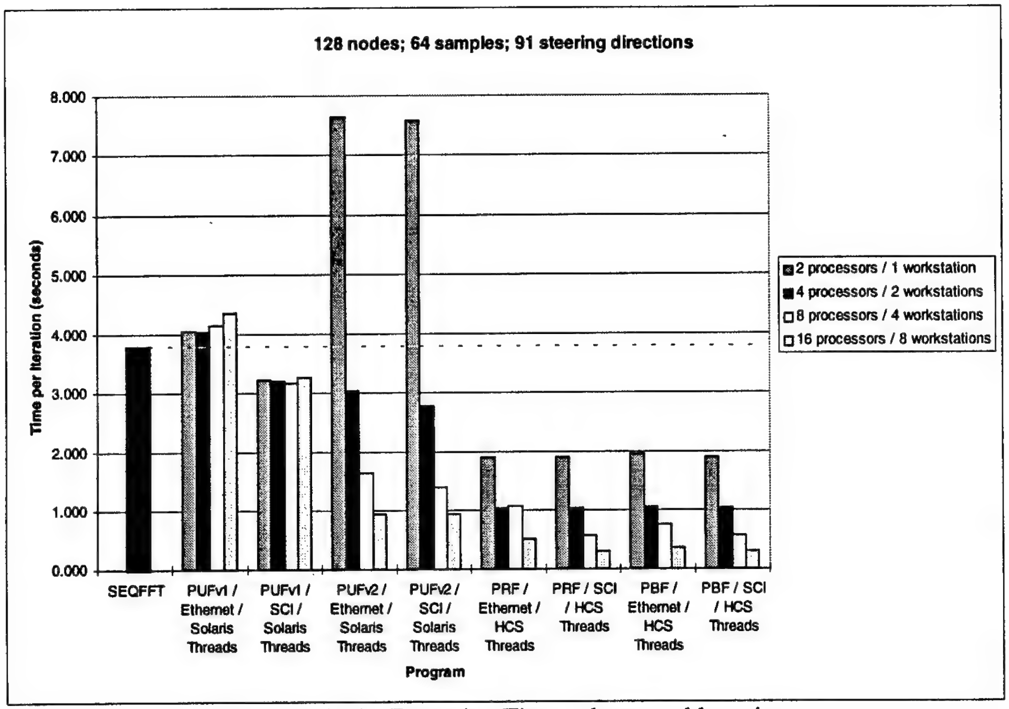


Figure 7.3.22 : Execution Times - large problem size

Execution times of all parallel programs and the purely sequential program (SEQFFT) for 128 nodes, 64 samples per FFT, and 91 steering directions.

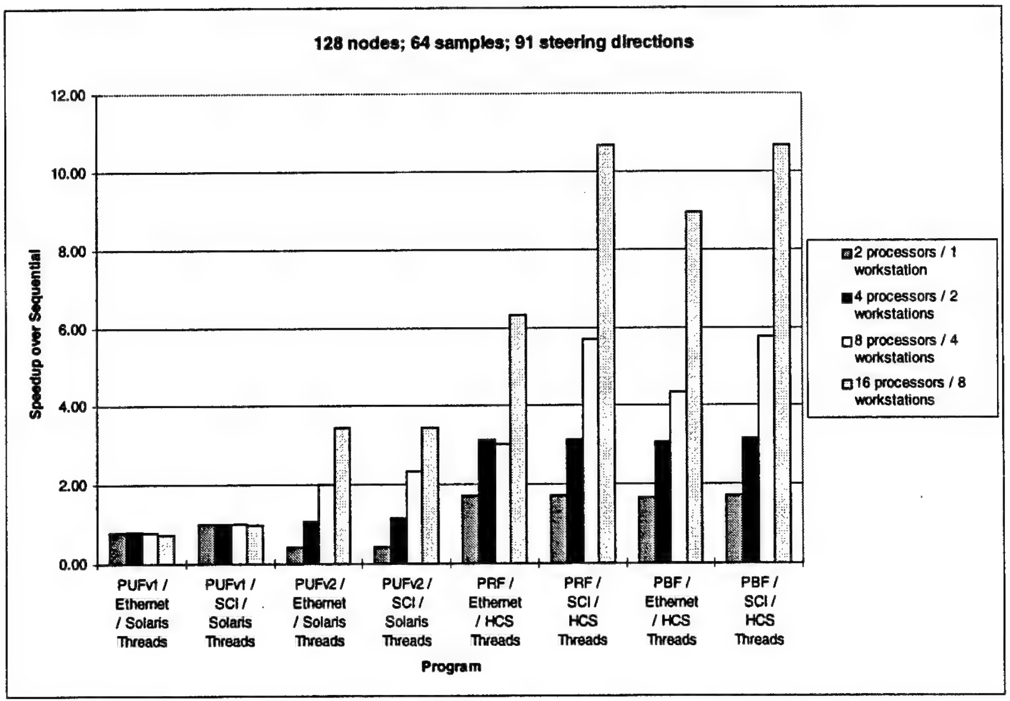


Figure 7.3.23 : Speedup vs. Pure Sequential - large problem size

Speedups of all parallel programs versus the purely sequential program (SEQFFT) for 128 nodes, 64 samples per FFT, and 91 steering directions.

The speedups show that the Ethernet versions of PRF and PBF outperform the sequential program by factors of between 6.3 and 8.5 for 16 processors in 8 workstations. The SCI versions outperform the sequential program by factors of about 11 on the same number of processors. Also of note is that PUFv2 has speedup of just under 4 in the large problem size, which is up from below 2 in the small problem size. This illustrates exactly how much the smaller amount of communication benefits PUFv2 over PUFv1.

The last two FFT beamforming algorithms, the ring and the bidirectional linear array, clearly perform well whether comparisons are made in the computer engineering realm or in the sonar array realm. The speedup is excellent over the purely sequential algorithms, which is good news for parallel programmers, and the speedup is excellent over the sonar array baseline, which is good news for sonar system designers.

8. Conclusions

In the first phase of this project, tasks have concentrated on the study, design, and analysis of the fundamental components in the design and analysis of parallel and distributed computing architectures and algorithms for fault-tolerant sonar arrays as well as models for their baseline counterparts. An interactive investigation of the interdependent areas of topology, architecture, protocol, and algorithms has been conducted. The results of this first phase have helped to identify the optimum candidate network topology, architecture, hardware components, and interface protocols for the system architecture and a set of fundamental decomposition techniques and parallel algorithms for a representative set of time-domain and frequency-domain beamforming methods.

The accomplishments for the first phase of this project can be summarized in terms of four research thrusts. First, an analysis of the effect of node outage on network reliability was conducted. Second, a survey of low-cost, low-power networking and processing components was started. When completed this survey will be used to determine the optimal network topology and system architecture. Third, fine-grain models of candidate network architectures have been developed for baseline, unidirectional ring, and bidirectional linear array topologies. Fourth, preliminary parallel decompositions of several standard beamforming algorithms have been performed, including delay-and-sum, delay-and-sum with interpolation, and FFT.

An analysis of the effects of random node failures on the beam power pattern has been completed. The biggest effect of node failure is in the side lobe pattern rather than in the main beam. The side lobe pattern is strongly dependent on the location of the failed nodes; however, the loss in the main beam power is independent of their location and depends only on the number of failed nodes. From the latest results it is apparent that a substantial number of nodes may fail before the main beam power is significantly degraded. Therefore it is essential to be able to bypass optically one or more failed nodes. The probability that the network will fail if the number of successive failed nodes exceeds the number that may be bypassed may be determined from simple combinatorial logic, and results suggest that it will be necessary to bypass three or more failed nodes if the network is to maintain connectivity while the array is still taking useable data. In anticipation of this requirement an SBIR Phase I award was made to Fiber and Sensor Technologies, Inc. for the development of a reliable, low-cost, low-loss micropowered bypass switch. A prototype piezoelectric switch developed in Phase I demonstrated that a final version to be developed in Phase II would be less than one cubic centimeter in size and dissipate less than 1 mW in the *through* state and no power in the *bypass* (fail-safe) state. The switch will be capable of bypassing 5 failed nodes. Estimated unit cost is about $25 in quantity.

A survey of low-cost, low-power networking and processing components is nearing completion. So far laser diodes, RAMs, analog-to-digital converters, and high-power batteries have been investigated and analyses of low-power clock recovery circuits, microprocessors, and plastic fiber optic cable are well underway. Initial results indicate that not only is it possible to design and implement the elements in this distributed processing sonar array via low-power custom devices, but based on cutting-edge technologies including Li/SO_2 batteries and low-power integrated circuits it may indeed be possible to attain the 30-day mission time requirement largely with COTS components. For example, processors are now emerging capable of less than 1 mA/MHz at 3V, low-power 3V DRAM devices will soon attain current drains as low as 9 mA for cycle times less than 1000 ns, and Li/SO_2 batteries now have an actual average capacity of over 200 Watt-hours/kilogram. Vertical cavity surface emitting lasers (VCSELs) are becoming available which offer sub-milliamp threshold currents. At present, they are expensive due to their low yield; however, they are the subject of extensive research due to their excellent temperature properties and it is likely that this will result in higher yields and lower cost in the future. Should these initial design estimates prove true in further experiments, this approach will dramatically increase the flexibility and versatility of the architecture, algorithms, and protocols under development allowing such arrays one day to support autonomously a wide variety of beamforming, detection, and classification mechanisms without changing the hardware.

Given that all technical issues are driven by the network topology design, the network protocol design, and the node processor, the development of an efficient and effective architecture and set of protocols for this network-based multicomputer system for autonomous sonar arrays represents a number of key technical challenges. Many important multicomputer architecture considerations must be addressed in terms of speed, cost, weight, power, and reliability for each node. To address these considerations, fine-grain network architecture models based on unidirectional linear array (i.e. the baseline), undirectional ring with register-insertion protocol, and bidirectional linear array have been completed, tested, and verified and a unidirectional ring with token-passing protocol is near completion. The models have and continue to be used for performance experiments and are being expanded to include fault-injection capability for dependability experiments as well.

The design and development of novel parallel and distributed algorithms for large sonar array beamforming applications, and the partitioning and mapping of these algorithms onto candidate network topologies, architectures, and protocols, is one of the most pivotal technical issues in this project. To attack this task in the project, a survey of all decomposition methods thus far in the field has been conducted from which a single comprehensive approach has been devised for this application area. By preparing a thorough, architecture-independent decomposition of candidate algorithms, the communication and computation requirements of each method can be thoroughly examined for application to each candidate architecture, at which time the most suitable parallel decomposed algorithms can be chosen for further experimentation. Meanwhile, a collection of conventional and non-conventional beamforming algorithms has been constructed. Preliminary parallel decompositions of several standard beamforming algorithms have been performed, including delay-and-sum, delay-and-sum with interpolation, and FFT. Algorithms and programs for the sequential version of these beamforming techniques have been developed in MATLAB and C sequential code to form a baseline by which true parallel algorithms and software are measured. Initial results from performance experiments with basic time-domain and frequency-domain parallel beamforming programs developed for this project indicate the potential for near-linear speedup and a high degree of parallel efficiency when properly mapped to network architectures similar to those candidates under consideration. A prototype X-Windows simulator has been constructed which supports multithreaded MPI parallel code which can be mapped to several networks, workstations, and processors in the parallel processing testbed developed for this project. Preliminary tests with this simulator have been successful in identifying and measuring the strengths and weaknesses of parallel programs being studied versus their baseline counterparts.

In the second phase of this research project (which began in January of 1997) tasks include the development of the preliminary software system for the advanced sonar array, the design of a strawman node circuit, and the development of a suite of simulation tools which support the design and analysis of both system performance and dependability. Parallel programs will be developed with inherent granularity knobs based on the results of algorithm decomposition and partitioning in the first phase. Self-healing extensions of the selected algorithms will be developed and simulated in this phase. A strawman hardware node design will be conducted to determine estimated gate counts and power requirements. Finally and concurrently, a suite of simulation tools will be developed and integrated to support the rapid virtual prototyping of topology, architecture, protocol, and algorithm selections made in the first phase. A revolutionary new interface between parallel beamforming programs written in MPI and the network and node architecture models constructed in BONeS is under development which promises to make it possible to conduct detailed performance analyses with cutting-edge parallel algorithms on candidate distributed sonar array architectures in a rapid virtual prototyping fashion. Several key elements of this second phase of the project (e.g. preliminary software for parallel delay-and-sum beamforming, preliminary software for parallel FFT beamforming, fundamental components for the system simulator) were begun ahead of schedule in order to better support the interactive nature of the tasks in the first and second phases.

The third phase of this project (which begins in January of 1998) involves the development, implementation, and demonstration of a small, laboratory-based hardware prototype with its own distributed, parallel, and embedded software system. This system will be used to verify and validate the simulation results, and in so doing demonstrate and better quantify the inherent advantages of the novel approach employed in its design. Critical factors in the evaluation of the system will center on quantitative measurements in the areas of performance and dependability, including computational speed, efficiency, and precision, reliability, cost, weight, power, size, and mission time, as well as qualitative measurements related to system flexibility, versatility, expandability, scalability, etc.

9. Literature Taxonomy

A taxonomy was created to categorize and organize the many beamforming techniques that were encountered during research. The delay-and-sum, delay-and-sum with interpolation, and FFT beamforming algorithms were studied in great detail because of their fundamental importance to understanding the basic issues in array signal processing. An alternative time-domain beamforming algorithm based on autocorrelation that was presented by Dr. Warren Rosen was also studied.

These fundamental algorithms were studied because many of the more advanced beamforming methods are extensions to the basic techniques. The advanced methods can be added as necessary because of the modular methods in which the programs were written. Likewise, the decomposition and implementation techniques used can easily be extended to the more complex analyses.

9.1 Conventional Techniques

Delay and Sum Beamforming	NEIL91, many others
Interpolation Beamforming	CHEN88, NEIL91
Beamforming by Autocorrelation	ROSE95
Frequency-Domain (FFT) Beamforming	CHEN88, DEFA88, NEIL91
Phase Shift Beamforming	CHEN88
Quadrature Beamforming	NEIL91
Time-Delay Bandpass Beamforming	HORV92
Capon's Minimum Variance Estimator	HAYK91, PILL89
Matched Field Processing	
Adapted Angular Response	
Linear Prediction Method	PILL89

9.2 Eigenvector-Based Techniques

PILL89

Completely Coherent Case

Symmetric Array Scheme: Coherent Sources in a Correlated Scene

Spatial Smoothing Schemes and Subarray Averaging

9.3 Adaptive Processing

Constrained Adaptation	DEFA88
Unconstrained Adaptation	DEFA88
Adaptive Mean-Level Detector	CHEN82
Adaptive-Combiner	NIEL91
Least-Mean-Square approach	HAYK91

Standard least-squares estimation	
Block optimization (sample matrix inversion algorithm)	
Recursive least-squares (RLS) algorithm	
Data decomposition-based least-squares estimation	
Rectangular orthogonalization (singular-value decomposition)	
Triangular orthogonalization (QR-decomposition)	
Linearly Constrained Minimum Variance (LCMV) Beamforming	
Minimum-Variance Distortionless Response (MVDR) Beamformers	
Generalized Eigenspace-Based Beamformers (GEIB)	YU95

9.4 Optimum Processing: Steady State Performance and the Wiener Solution

Minimum Mean Square Error (MMSE)	PILL89
Maximization of Signal-to-Noise Ratio (MSNR)	PILL89

9.5 Matched Field Processing (MFP) BAGG93

Conventional MFP	
Power Law MFP	
Minimum Variance, Distortionless Filter MFP (MVDF)	HAYK91
Controlled Sensor Noise MVDF	
Matched Mode Processing	
Subarray Methods	
Minmax, Prony	

9.6 Maximum Likelihood (ML)

Explicit Approximation ML	HERT95
Stochastic model of waveforms	VIBE95
Deterministic model of waveforms	VIBE95

9.7 Time Delay Estimation

Multipath Cancellation Time Delay Estimation (MCTDE)	CHIN94
Multipath Equalization Time Delay Estimation (METDE)	CHIN94
Recursive Registration	NAMA94
Generalized Maximum Likelihood	NAMA94
Minimum Mean Squared Error	NAMA94

9.8 Direction of Arrival Estimation (DOA)

Multiple Signal Classification (MUSIC)	HAYK95
Unknown Noise-MUSIC (UN-MUSIC)	HAYK95
Unknown Noise-Correlation and Location Estimation (UN-CLE)	HAYK95
Minimum Norm Estimation	HAYK95, ERMO94
Linearly Constrained Minimum Variance	
MVDR	HAYK91

9.9 Miscellaneous

Optimum Time using Simulated Annealing	COLL93
Fractional Beamforming	COLL93
Parametric Estimation (PARADE)	HAYK95
Augmentation Technique	PILL89
Estimation of Signal Parameters via Rotational Invariance Techniques (ESPRIT)	
Total Least Squares ESPRIT (TLS-ESPRIT)	
Generalized Eigenvalues Utilizing Signal Subspace Eigenvectors (GEESE)	
Radial-basis Function Associative Memory	
Direction Finding Using First Order Statistics	PILL89
Phase-locked Loop (PLL)	HAYK95
Shape Calibration	FUCH95

10. Bibliography

10.1 Battery Technology

[BRO90] Bro, P. and S. C. Levy. *Quality and Reliability Methods for Primary Batteries.* New York: Wiley & Sons, 1990.

[LIND95] Linden, David, ed. *Handbook of Batteries.* New York: McGraw-Hill, 1995.

[TUCK91] Tuck, Clive D. S. *Modern Battery Technology.* New York: Ellis Horwood, 1991.

10.2 Beamforming, General

[BIEN77] Bienvenu, G. "An Adaptive Approach to Underwater Passive Detection." *Aspects of Signal Processing, Part 1.* Ed. G. Tacconi. Dordrecht, Holland: D. Reidel, 1977: 395-400.

[BURD91] Burdic, William S. *Underwater Acoustic System Analysis.* 2nd ed. New Jersey: Prentice Hall, 1991.

[CAND94] Candy, J. V. and E. J. Sullivan. "Model-Based Processing for a Large Aperture Array." *IEEE Transactions on Oceanic Engineering* 19.4 (Oct. 1994): 519-28.

[CHEN82] Chen, C. H. *Digital Waveform Processing and Recognition.* Boca Raton: CRC Press, 1982.

[CHEN88] Chen, C. H. ed. *Signal Processing Handbook.* New York: Marcel Dekker, 1988.

[CHIN94] Ching, P. C. and H. C. So. "Two Adaptive Algorithms for Multipath Time Delay Estimation." *IEEE Transactions on Oceanic Engineering* 19.3 (Jul. 1994): 458-62.

[CROC81] Crochiere, Ronald E. and Lawrence R. Rabiner. "Interpolation and Decimation of Digital Signals - A Tutorial Review." *Proceedings of the IEEE* 69.3 (Mar. 1981): 300-31.

[DEFA88] DeFatta, David J., Joseph G. Lucas, and William S. Hodgkiss. *Digital Signal Processing: A System Design Approach.* New York: Wiley, 1988.

[ERMO94] Ermolaev, Victor T. and Alex B. Gershman. "Fast Algorithm for Minimum-Norm Direction-of-Arrival Estimation." *IEEE Transactions on Signal Processing* 42.9 (Sep. 1994): 2389-93.

[FARR77] Farrier, D. R. and T. S. Durrani. "Signal Extraction Algorithms for Adaptive Processing of Array Data." *Aspects of Signal Processing Part 2*. Ed. G. Tacconi. Dordrecht, Holland: D. Reidell, 1977: 485-94.

[FUCH95] Fuchs, Jean-Jacques. "Shape Calibration for a Nominally Linear Equispaced Array." *IEEE Transactions on Signal Processing* 43.10 (Oct. 1995): 2241-8.

[GERS95] Gershman, Alex B., Victor I. Turchin, and Vitaly A. Zverev. "Experimental Results of Localization of Moving Underwater Signal by Adaptive Beamforming." *IEEE Transactions on Signal Processing* 43.10, (Oct. 1995): 2249-57.

[GOODa] Goodwin, Michael, and Gary Elko. "Constant Beamwidth Beamforming Using an Affine Phase Multi-Beamformer." Murray Hill, New Jersey: AT&T Bell Laboratories, Acoustics Research Department. http://ptolemy.eecs.berkeley.edu/~michaelg/constant.ps .

[GOODb] Goodwin, Michael. "Frequency-Independent Beamforming." Berkeley, CA: U of Berkeley, EECS, and AT&T Bell Laboratories, Acoustics Research. http://ptolemy.eecs.berkeley.edu/~michaelg/fib.ps .

[GORI] Goris, Malcolm J., and Donald J. Mclean. "Towed Array-Shape Estimation: A Comparison of Methods." CSIRO, Division of Radiophysics. http://www.rp.csiro.au/people/mgoris/papers/asecomp.ps .

[GRIF77] Griffiths, J. W. R., and J. E. Hudson. "An Introduction to Adaptive Processing in a Passive Sonar System." *Aspects of Signal Processing, Part 1.* Ed. G. Tacconi. Dordrecht, Holland: D. Reidel, 1977: 299-308.

[GRIF73] Griffiths, J. W. R., P. L. Stoklin, and C. van Schooneveld, eds. *Signal Processing; Proceedings*. Academic Press: London, 1973.

[HAMP93] Hampson, Grant and Andrew P. Papliński. "Beamforming by Interpolation." Technical Report 93/12. Clayton, Victoria, Australia: Monash U, 1993. ftp://ftp.rdt.monash.edu.au/pub/techreports/RDT/93-12.ps.Z .

[HAMP95] Hampson, Grant and Andrew P. Papliński. "Simulation of Beamforming Techniques for the Linear Array of Transducers." Technical Report 95/3. Clayton, Victoria, Australia: Monash U, 1995. ftp://ftp.rdt.monash.edu.au/pub/techreports/RDT/95-3.ps.Z .

[HAYK91] Haykin, Simon. *Adaptive Filter Theory.* 2nd ed. Englewood Cliffs: Prentice Hall, 1991.

[HAYK95] Haykin, Simon, ed. *Advances in Spectrum Analysis and Array Processing Volume III.* Englewood Cliffs: Prentice Hall, 1995.

[HOLM] Holm, Sverre. "Digital Beamforming in Ultrasound Imaging." Oslo, Norway: U of Oslo, Department of Informatics. ftp://ftp.ifi.uio.no/pub/publications/others/SHolm-4.ps.Z .

[HORV92] Horvat, Dion C. M., John S. Bird, and Martie M. Goulding. "True Time-Delay Bandpass Beamforming." *IEEE Transactions Oceanic Engineering* 17.2 (Apr. 1992): 185-92.

[IFEA93] Ifeachor, Emmanuel C. *Digital Signal Processing: A Practical Approach.* Workingham, England: Addison-Wesley, 1993.

[JOHN93] Johnson, Don H. and Dan E. Dudgeon. *Array Signal Processing, Concepts and Techniques.* New Jersey: Prentice Hall, 1993.

[LI95] Li, Shaolin and Terrence J. Sejnowski. "Adaptive Separation of Mixed Broad-Band Sound Sources with Delays by a Beamforming Hérault-Jutten Network." *IEEE Transactions on Oceanic Engineering* 20.1 (Jan. 1995): 73-8.

[LUND77] Lunde, E. B. "The Forgotten Algorithm in Adaptive Beamforming." *Aspects of Signal Processing, Part* 2. Ed. G. Tacconi. Dordrecht, Holland: D. Reidel, 1977: 411-21.

[MCWH89] McWhirter, J.G and T.J. Shepherd. "Systolic Array Processor for MVDR Beamforming," *IEE Proceedings* 136.F2 (Apr. 1989):75-80.

[MORG94] Morgan, Don. *Practical DSP Modeling, Techniques, and Programming in C.* New York: Wiley, 1994.

[NIEL91] Nielsen, Richard O. *Sonar Signal Processing.* Boston: Artech House, 1991.

[NORD94] Nordebo, Sven, Ingvar Claesson, and Sven Nordholm. "Adaptive Beamforming: Spatial Filter Designed Blocking Matrix." *IEEE Transactions on Oceanic Engineering* 19.4 (Oct. 1994): 583-9.

[PILL89] Pillai, Unnikrishna S. *Array Signal Processing.* New York: Springer-Verlag., 1989.

[SMIT95] Smith, Winthrop W. and Joanne M. Smith. *Handbook of Real-Time Fast Fourier Transforms: Algorithms to Product Testing.* New York: IEEE, 1995.

[TANG94] Tang, C. F. T., K. J. R. Liu, and S. A. Tretter. "Optimal Weight Extraction for Adaptive Beamforming Using Systolic Arrays," *IEEE Transactions on Aerospace and Electronics Systems* 30.2 (Apr. 1994): 367-84.

[TRAN93] Tran, Jean-Marie Q. D. and William S. Hodgkiss. "Spatial Smoothing and Minimum Variance Beamforming on Data from Large Aperture Vertical Line Arrays." *IEEE Transactions Oceanic Engineering* 18.1 (Jan. 1993): 15-23.

[VANP95] Vanpoucke, Filiep and Marc Moonen. "Systolic Robust Adaptive Beamforming with an Adjustable Constraint," *IEEE Transactions on Aerospace and Electronic Systems* 31.2 (Apr. 1995): 658-68.

[VIBE95] Viberg, Mats, Björn Ottersten, and Arye Nehorai. "Performance Analysis of Direction Finding with Large Arrays and Finite Data." *IEEE Transactions on Signal Processing* 43.2 (Feb. 1995): 469-76.

[YU95] Yu, Jung-Lang and Chien-Chung Yeh. "Generalized Eigenspace-Based Beamformers." *IEEE Transactions on Signal Processing* 43.11 (Nov. 1995): 2453-61.

[ZVAR93] Zvara, George P. "Real Time Time-Frequency Active Sonar Processing: A SIMD Approach," *IEEE Journal of Oceanic Engineering* 18.4 (Oct. 1993): 520-8.

10.3 Hardware, General

[ARM96] "ARM6 and ARM7 Cores." Technical Manual, Advanced RISC Machines, Ltd., 1996. http://www.arm.com/Pro+Peripherals/Cores/ARM6+7/ .

[KAJI97] Kajita, Mikihiro et al. "1-Gbps Modulation Characteristics of a Vertical Cavity Surface-Emitting Laser Array Module." *Photonic Technology Letter* 9.2 (Feb. 1997).

[MAYE97] Mayer, John H. "16-bit MCUs Take On Complex Embedded Applications." *Computer Design.* Jan. 1997: 116-20.

10.4 Low-Power Analog-to-Digital Converters

[CHOa] Cho, Thomas B. and Paul R. Gray. "A 10-bit, 20-MS/s, 35-mW Pipeline A/D Converter." U. of California, Berkeley. http://kabuki.eecs.berkeley.edu/papers.html#adc .

[CHOb] Cho, Thomas, et. al. "Design Considerations for High-Speed Low-Power." U. of California, Berkeley. http://kabuki.eecs.berkeley.edu/papers.html#adc .

[HORO95] Horowitz, Paul and Winfield Hill. *The Art of Electronics.* New York: Cambridge U. Press, 1995.

[HOES94] Hoeschelle, David F. *Analog-to-Digital and Digital-to-Analog Conversion Techniques.* New York: Wiley, 1994.

[KING94] King, Eric et. al. "Parallel Delta-Sigma A/D Conversion." *IEEE Custom Integrated Circuits Conference*, 1994: 503-6.

[KUSU93] Kusumoto, Keiichi and Akira Matsuzawa. "A 10-bit 20-MHz 30-mW Pipelined Interpolating CMOS ADC." *IEEE Journal of Solid-State Circuits* 28.12 (Dec. 1993): 1200-6.

[NAKA95] Nakamura, Katsufumi, et. al. "An 85 mW, 10 b, 40 Msample/s CMOS Parallel-Pipelined ADC." *IEEE Journal of Solid-State Circuits* 30.3 (Mar. 1995): 173-83.

[RAZA95] Razavi, Behzad. *Principles of Data Conversion System Design.* New Jersey: IEEE Press, 1995.

[SATO94] Satou, Kouichi, et. al. "A 12 Bit 1MHz ADC with 1mW Power Consumption." *IEEE Custom Integrated Circuits Conference*, 1994: 515-8.

[SONG95] Song, Won-Chul, et. al. "A 10-b 20-Msample/s Low-Power CMOS ADC." *IEEE Journal of Solid-State Circuits* 30.5 (May 1995): 514-21.

[VAN94] van de Plassche, Rudy. *Integrated Analog-to-Digital and Digital-to-Analog Converters.* Dordecht: Kluwer Academic, 1994.

[YAMA94] Yamamura, Ken, Akihiko Nogi, and Allen Barlow. "A Low-Power 20 Bit Instrumentation Delta-Sigma ADC." *IEEE Custom Integrated Circuits Conference*, 1994: 519-22.

10.5 Low-Power RAM

[4MEG95] "4 MEG DRAM Typical Operating Curves." Technical Note TN-04-23, Micron Technology, Inc., 1995. http://www.micron.com/ .

[256K95] "256K x 16 DRAM Typical Operating Curves." Technical Note TN-04-22, Micron Technology, Inc., 1995. http://www.micron.com/ .

[DESI95a] "A Designer's Guide to 3.3V SRAMs." Technical Note TN-05-16, Micron Technology, Inc., 1995. http://www.micron.com/ .

[DESI95b] "Designing for SmartVoltage Technology Flash." *Design Line* 4.4, Micron Technology, Inc., 1995. http://www.micron.com/ .

[DRAM95] "DRAM Considerations for PC Memory Design." Technical Note TN-04-15, Micron Technology, Inc., 1995. http://www.micron.com/ .

[ITOH95] Itoh, Kiyoo, et. al. "Trends in Low-Power RAM Circuit Technologies." *Proceedings of the IEEE* 83.4 (Apr. 1995): 524-43.

[LOWP95] "Low-Power Memory Design Using Data Retention." Technical Note TN-05-17, Micron Technology, Inc., 1995. http://www.micron.com/ .

[SRAM95] "SRAMs and Low-Voltage Data Retention." Technical Note TN-05-19, Micron Technology, Inc., 1995. http://www.micron.com/ .

[USIN95] "Using 3.3V DRAMs in 5V Applications." Technical Note TN-04-34, Micron Technology, Inc., 1995. http://www.micron.com/ .

[VARI94] "Various Methods of DRAM Refresh." *Design Line* 3.3, Micron Technology, Inc., 1994. http://www.micron.com/ .

10.6 Networking Background

[BERT92] Bertsekas, Dimitri and Robert Gallager. *Data Networks.* New Jersey: Prentice Hall, 1992.

[FISH95] Fishwick, Paul A. *Simulation Model Design and Execution, Building Digital Worlds.* New Jersey: Prentice Hall, 1995.

[HAMM88] Hammond, Joseph L. and Peter J. P. O'Reilly. *Performance Analysis of Local Computer Networks.* Massachusetts: Addison Wesley, 1988.

[SPUR95] Spurgeon, Charles. *Guide to 10-Mbps Ethernet.* Austin, Texas: U. of Texas, Austin Networking Services, 1995.

[STAL94] Stallings, William. *Data and Computer Communications.* New York: Macmillan, 1994.

10.7 Parallel Computing

[AMDA67] Amdahl, G. M. "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities." *Proc. AFIPS* 30 (1967): 483-5.

[BAL90] Bal, H.E. *Programming Distributed Systems.* New Jersey: Silicon Press, 1990.

[BECK96] Beck, Alan, ed. "Dave Gustavson Answers Questions About SCI," *HPCWire*, Oct. 4 1996.

[CARR90] Carriero, Nicholas and David Gelernter. *How to Write Parallel Programs: A First Course.* Cambridge: MIT Press, 1990.

[CORM95] Cormen, Thomas H. and Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms.* New York: McGraw-Hill, 1995.

[FLYN72] Flynn, M.J. "Some Computer Organizations and Their Effectiveness." *IEEE Trans. Computers* 21.9 (1972): 948-60.

[FORE96] *ForeRunner™ SBA-200 ATM Sbus Adapter User's Manual, MANU0069,* Version 4.0, FORE Systems, Inc., Rev. A-March 1996.

[FORT78] Fortune, S. and J. Willie. "Parallelism in Random Access Machines." *Proc. 10th Annual ACM Symp. on Theory of Computing* (1978): 114-8.

[FOST95] Foster, Ian. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Reading, Mass: Addison-Wesley, 1995.

[GIBB88] Gibbons, Alan and Wojciech Rytter. *Efficient Parallel Algorithms.* Cambridge: Cambridge U. Press, 1988.

[GROP] Gropp, William, Ewing Lusk, Nathan Doss, and Anthony Skjellum. "A High-performance, Portable Implementation of the MPI Message-Passing Interface Standard."

[GROP95] Gropp, William and Ewing Lusk and Anthony Skjellum. *Using MPI, Portable Parallel Programming with the Message-Passing Interface.* Massachusetts: MIT Press, 1995.

[GROP96] Gropp, William and Ewing Lusk. *User's Guide for mpich, a Portable Implementation of MPI,* Chicago: U. of Chicago, 1996.

[GUST88] Gustafson, J. L. "Reevaluating Amdahl's Law." *Commun. ACM* 31.5 (May 1998): 532-3.

[HWAN93] Hwang, Kai. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* New York: McGraw-Hill, 1993.

[LADD95] Ladd, Scott Robert. *C++ Templates and Tools.* New York: M&T Books, 1995.

[LEWI92] Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing.* New Jersey: Prentice-Hall, 1992.

[LIN83] Lin, Shu and Daniel J. Costello, Jr. *Error Control Coding: Fundamentals and Applications.* New Jersey: Prentice Hall, 1983.

[MESS94] Message-Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* Knoxville: U. of Tennessee, May 1994.

[PAIG93] Paige, Robert, John Reif, and Ralph Wachter, eds. *Parallel Algorithm Derivation and Program Transformation.* Boston: Kluwer, 1993.

[NICH96] Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming.* Cambridge: O'Reilly, 1996.

[PROG95] "Programmer's Guide to MPI for Dolphin's Sbus-to-SCI Adapters Version 1.0." U. of Bergen: Parallab, Nov 1995.

[QUIN94] Quinn, Michael J. *Parallel Computing: Theory and Practice.* New York: McGraw-Hill, 1994.

[RAGS91] Ragsdale, Susann, ed. *Parallel Programming.* New York: McGraw-Hill, 1991.

[SNIR96] Snir, Marc, S. et. al. *MPI: The Complete Reference.* Cambridge: MIT Press, 1996.

[TAKE92] Takeuchi, Akikazu. *Parallel Logic Programming.* New York: Wiley, 1992.

[ZOMA96] Zomaya, Albert, ed. *Parallel and Distributed Computing Handbook.* New York: McGraw-Hill, 1996.

10.8 UNIX Programming

[ROBB96] Robbins, Kay A. and Steven Robbins. *Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading.* Upper Saddle River, N.J: Prentice Hall, 1996.

[STEV92] Stevens, W. Richard. *Advanced Programming in the UNIX Environment.* Massachusetts: Addison Wesley, 1992.

Appendix A: Altia Overview

A. Altia Overview

The Altia Graphics package defines many animated controls which may be pasted into a screen. The files *gui.dsn*, *instrmnt.dsn*, and *menus.dsn* have many animations including rotary knobs, slide controls, text and numerical I/O boxes. To paste them into a new file, the user simply has to drag the animation into the layout. In addition, Altia provides other useful drawing tools usually featured in powerful graphic painting programs. This way a graphical user interface (GUI) can resemble a physical device such as a radio communicator or the front end of a rack mounted module.

The Altia_Lite package only supports a few of the many features of the full version of Altia such as geometric shapes, lines, boxes and circles. Also many fonts are included which can be used to identify controls. Graphics are all based on discrete shapes which may be blocked together which allows for flexibility and powerful editing capabilities.

Another powerful feature is the use of "cards" which can map different layers of GUI interfaces together in one program. A control such as a rotary knob can choose the page or card which the user is currently using. This is particularly useful if there are too many parameters to be entered on one page or if there are multiple modes which need to be supported. For example, the future FTSA GUI will have a card for each algorithm so that each algorithm can support its own discrete parameter list without having to build GUIs for each one.

The animated controls coordinate with the background program interpreter through named parameters. Each device must have its own set of named parameters to coordinate successfully. The background program, which interprets the actions of the GUI, calls functions with the unique controller names. The Altia Library is the set of functions which can be integrated into C/C++ which provides various utilities and functions for the interpreter program. A few of these include *AtSendText*(), *AtSendEvent()*, *AtGetText()*, *AtPollEvent()*, *AtAddCallback()*, and *AtMainLoop()*. The first four functions are used to retrieve text or an event from the GUI or send text or an event to the GUI. An event is defined as a change in state of some device such as a toggle or a multi-position switch. Text boxes may be used to pass integral parameters to and from the GUI. An example of this is a status indicator text box showing the status of an operation such as "Please Wait" or "Computing". The other two functions *AtAddCallback()* and *AtMainLoop()* are the control functions. A device such as a toggle switch, represented by an event, may be tied to a function call with the use of the *AtAddCallback()* function in combination with *AtMainLoop()*. For example, this portion of code,

```
AtAddCallback(connectId,"push_test_af",Test_AF,NULL);
AtAddCallback(connectId,"push_find_source",Find_Source,NULL);
AtAddCallback(connectId,"push_save",Save_Settings,NULL);
AtAddCallback(connectId,"power_toggle",Shut_Down,NULL);
AtAddCallback(connectId,"push_load",Load_Settings,NULL);
AtAddCallback(connectId,"push_defaults",Load_Defaults,NULL);

AtMainLoop();
```

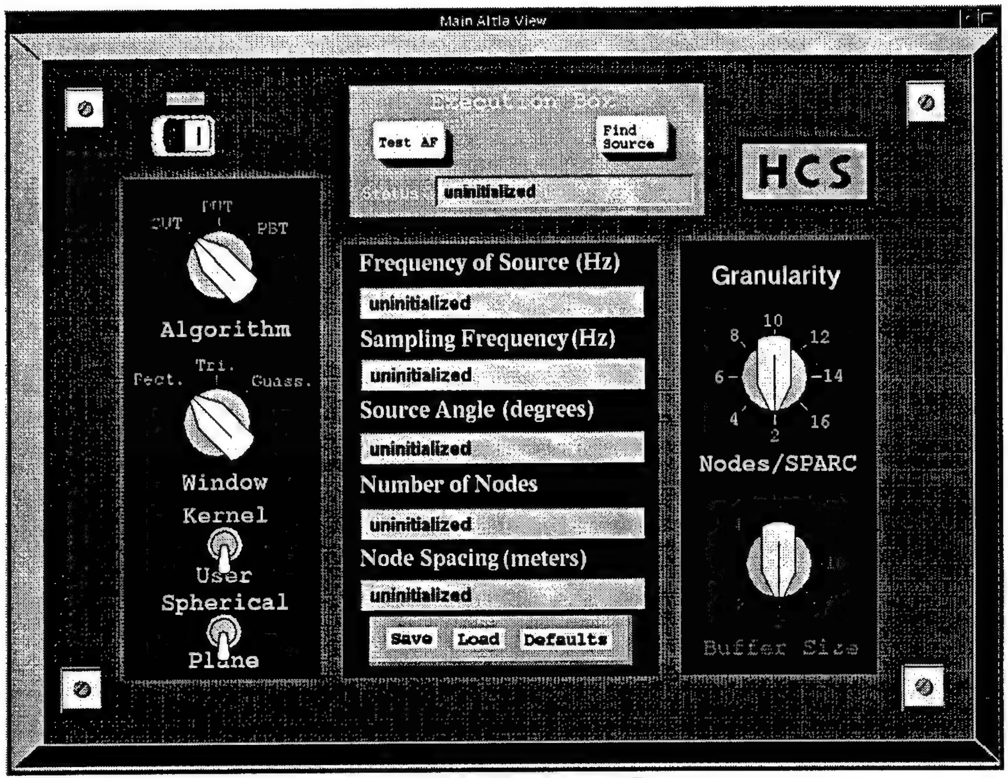
polls the six event sources which are all push buttons, except the *power_toggle* which is a toggle switch. *AtMainLoop()* monitors the state of each of these devices and calls the function associated with the event if the state is changed. Pushing the button Load, which is referenced by the parameter name *push_load*, calls the subroutine *Load_Settings*.

To compile and link the GUI to the interpreter program requires compiling the source code with the library files. For example the compile line,

```
gcc -o gui ftsagui.c time_keeper.o -lm /hdisk2/Nuthena41/altia_lite/lib/liblan.a -lsocket -lnsl
```

will link the functions previously discussed into the C executable, *gui**. Running the program requires the GUI and the interpreter to communicate via a socket. A socket is a communication channel used in UNIX based systems for interprocessor communication.

Currently, the features of the GUI shown in Figure A.1 allow the user to execute SUT and PUT simulations. This will soon include PRT, PBT, SUF, PUF, PRF, and PBF. The code for the coordinating interface program *ftsagui.c* is shown in the source code section of this appendix. It interprets the actions of the user and calls functions to execute the operation or action. Specific array and run parameters may be set by using intuitive controls (i.e. granularity knobs). Upon completion of a simulation run, a status indicator displays the total elapsed runtime. The GUI also allows the user to save, load settings, and load default settings. The GUI allows simultaneously polling of any knob or button which is coordinated by the *AtMainLoop* Altia function previously discussed. The execution box contains buttons to spawn off the MPI-threaded beamform algorithms. It also holds the status indicator to inform the user of what actions are currently taking place. For instance, when running a simulation the status indicator presents the message "Computing...Please Stand By". When operating the GUI push buttons, the interpreter polls all of the function knobs and the parameter list and sets up a long list of arguments which are passed to the appropriate executable program. The algorithm writes its results to a file which may be used for data analysis. Once this file is written, the GUI interpreter spawns off an *xterm* with the program *go_gnu*. This runs a batchfile to plot the results onto the screen. The system call to *go_gnu* sets a few parameters of UNIX Gnuplot and plots the elements of the output file.



Status: uninitialized

Frequency of Source (Hz)
uninitialized

Sampling Frequency (Hz)
uninitialized

Source Angle (degrees)
uninitialized

Number of Nodes
uninitialized

Node Spacing (meters)
uninitialized

Figure A.1 : Altia GUI Screen Capture

The screen capture shows the following controls (counter-clockwise from top-left): power, algorithm, windowing function, thread level, spherical spreading, buffer size, and sonar nodes per SPARC processor.

An alternate GUI used to observe the results of BONeS simulations in real-time is shown below as Figure A.2. Currently, the GUI is supported through UNIX file sharing and the use of the construct "poll()". Future BONeS implementations will interface with the Altia libraries via shared memory, namely pipes, FIFO's, or other UNIX interprocessor communication techniques. The *Altiaportal* block models in BONeS open files so that integer data values may be written to the temporary files to be displayed onto the GUI. The file *poll.c* shown in the source code section is a daemon process which constantly polls the files and updates the GUI interface routine when any changes are made. The process which communicates with the GUI is called *altiaportal.c* and is also shown in the source code section.

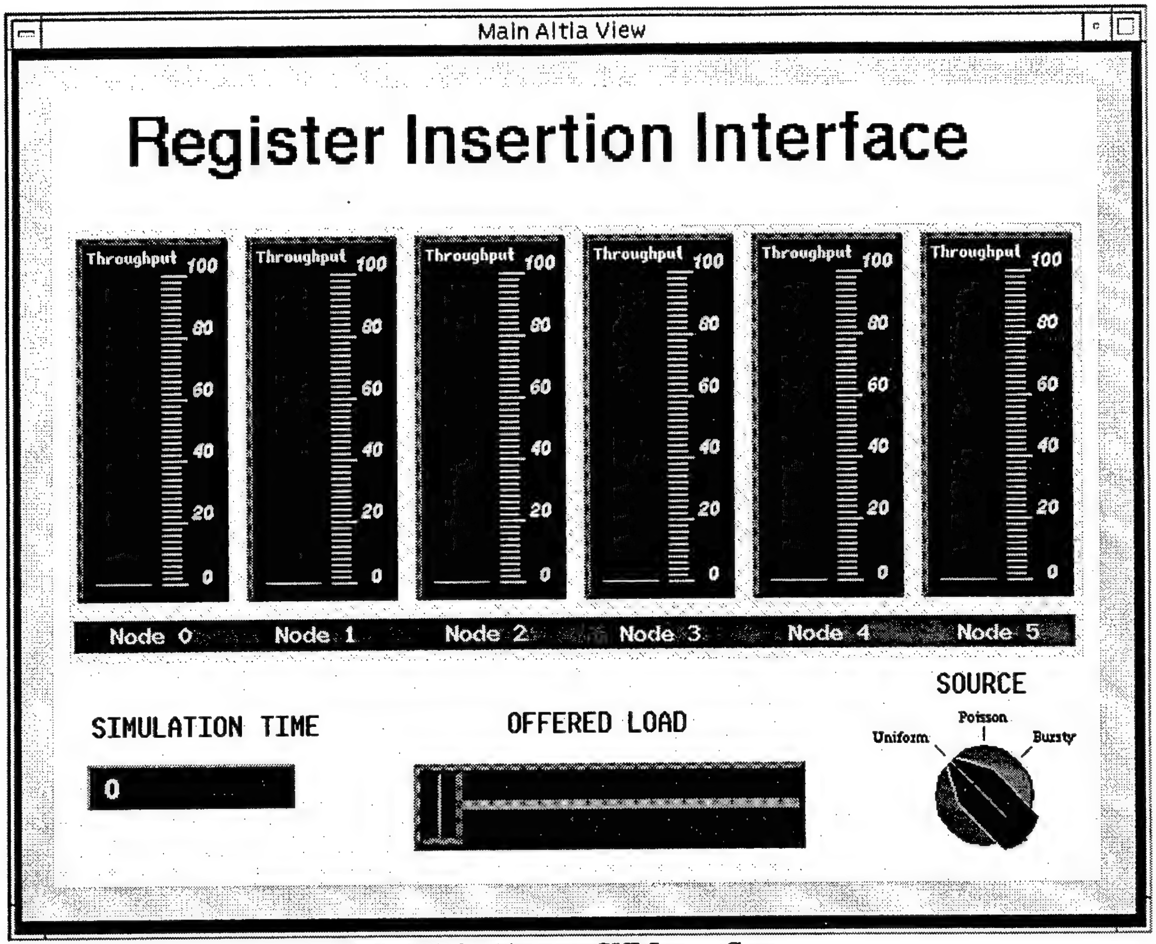


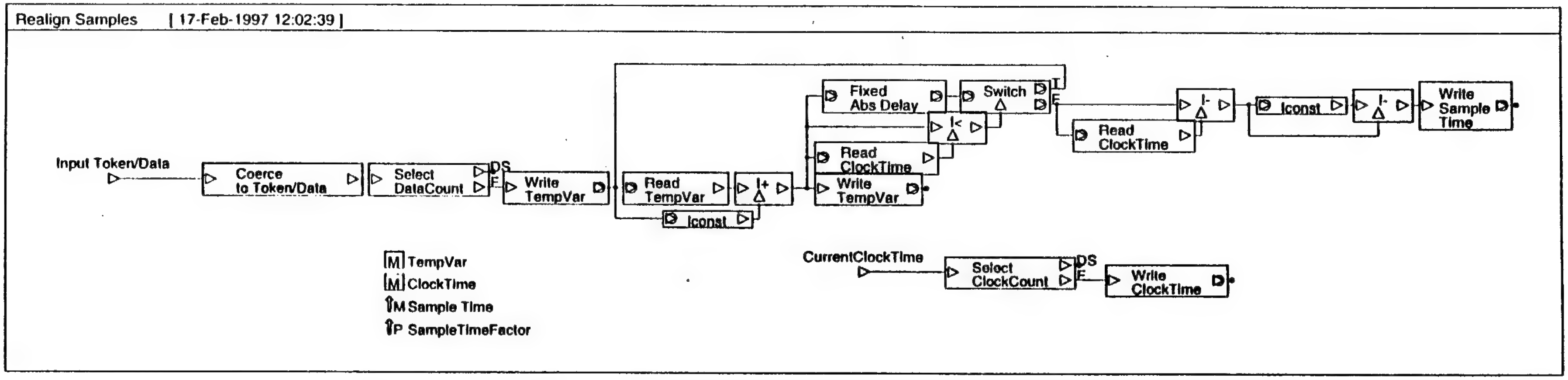
Figure A.2 : Alternate GUI Screen Capture

A screen shot from the alternate GUI used for observing BONeS simulation results in real-time.

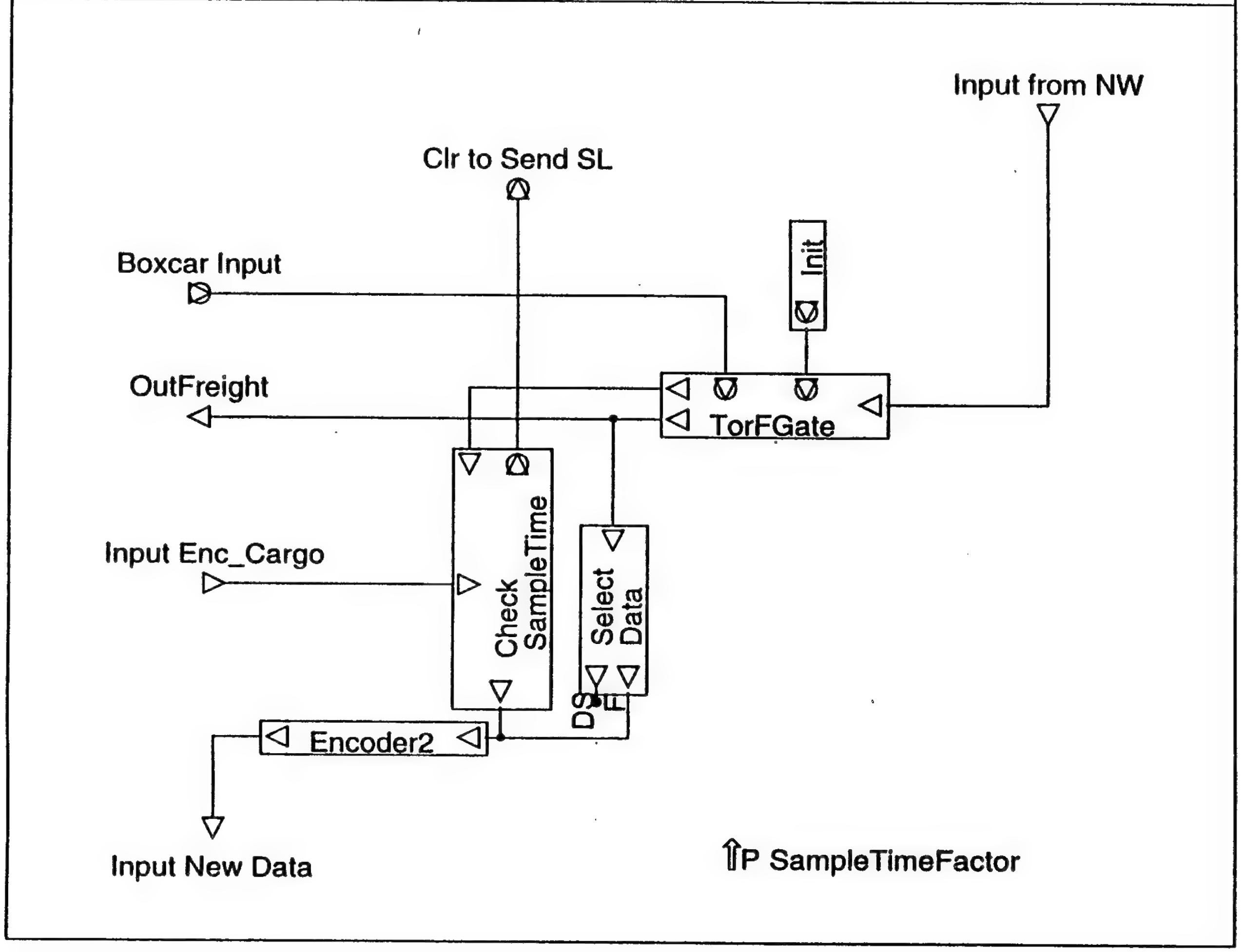
Overall, the Altia GUI development tool allows an easy and useful GUI to be created to ease the use of programs with complex input requirements. Since the Altia front end operates in real-time, the parameters of a program can be changed during execution or the status of an execution can be shown graphically. With the proliferation of graphical windowing environments for workstations and personal computers, the Altia tool allows rapid user interfacing for programs that would instead use arcane command line parameters or configuration files. This keeps new programs as visually stimulating and user friendly as the rest of the operating system.

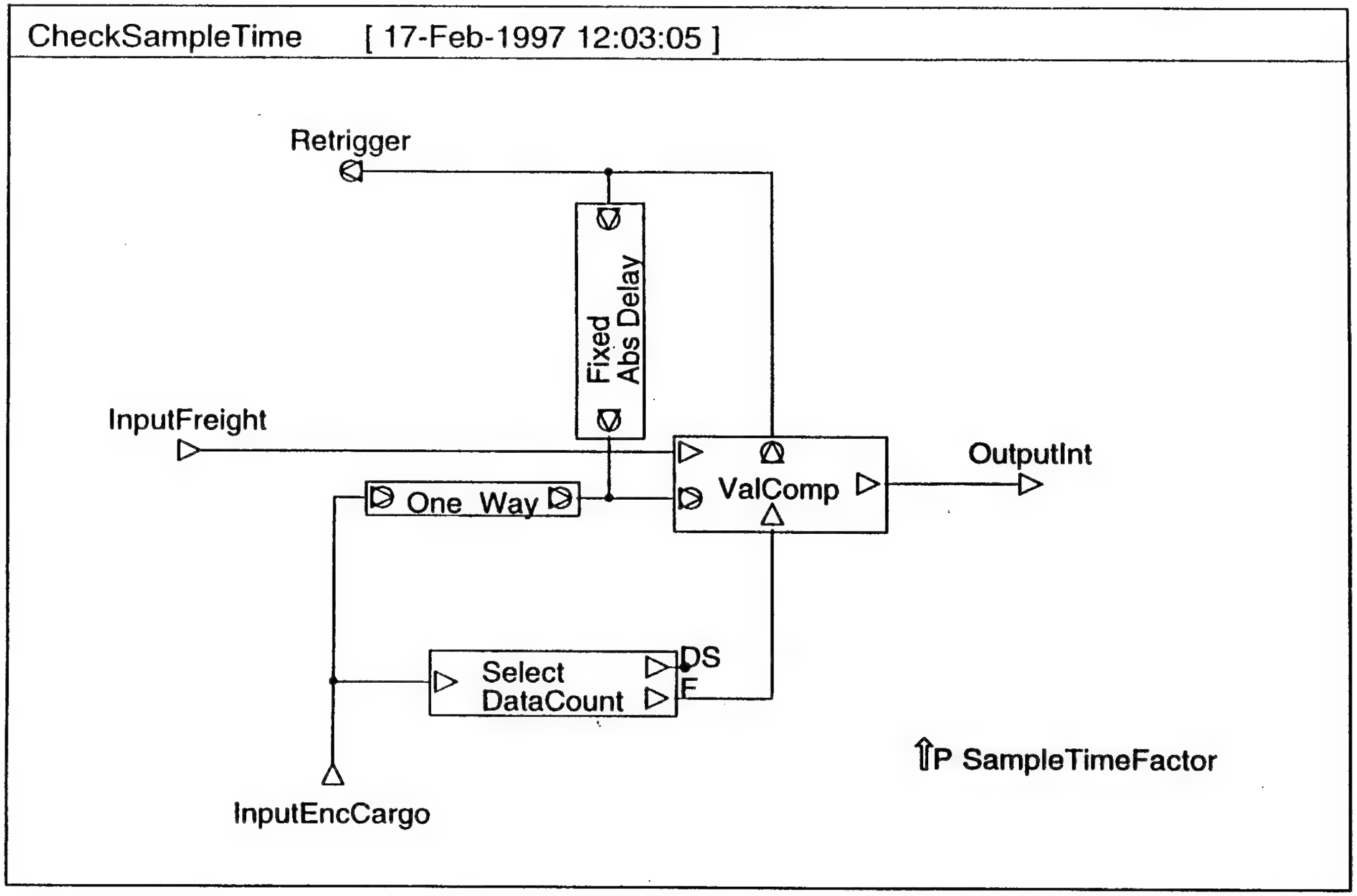
Appendix B: BONeS Models

Realign Samples [17-Feb-1997 12:02:39]

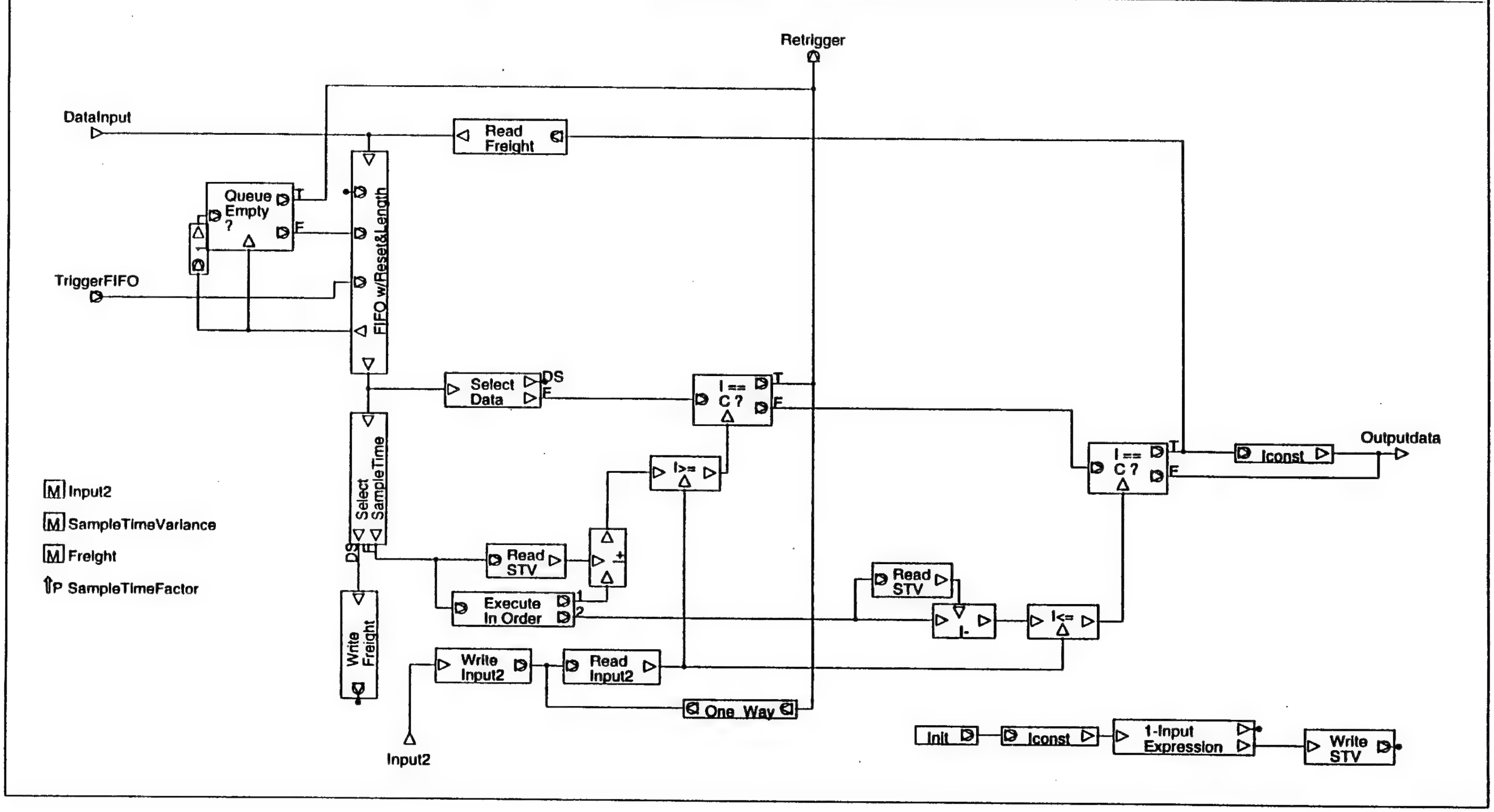


DataCheck [17-Feb-1997 12:02:57]

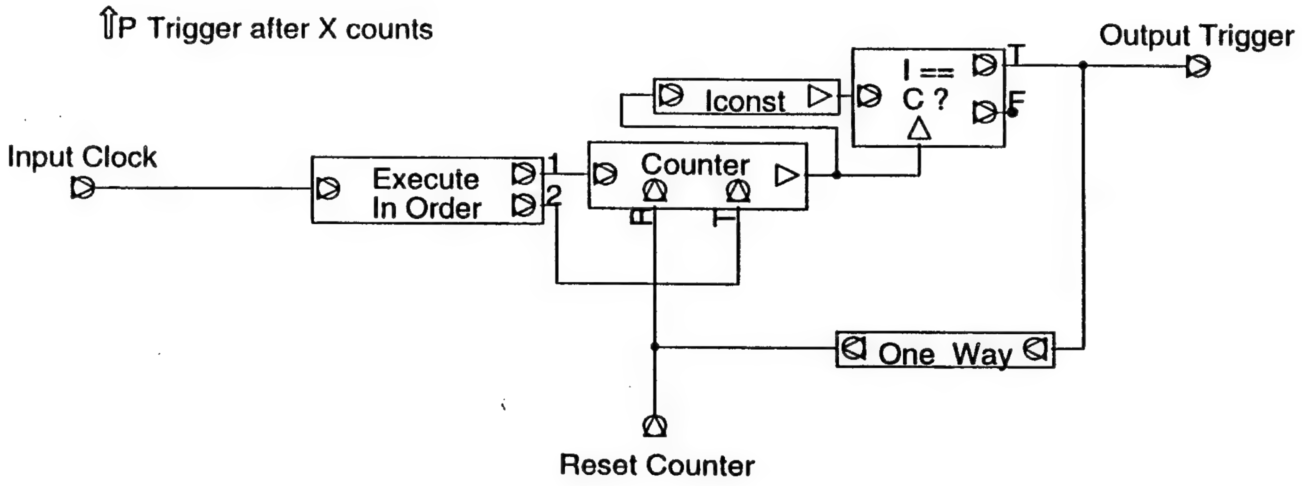



CheckSampleTime
[17-Feb-1997 12:03:05]
Retrigger
Fixed
Abs Delay
InputFreight
One Way
ValComp
OutputInt
Select
DataCount
DS
F
InputEncCargo
⇧P SampleTimeFactor

ValComp [17-Feb-1997 12:03:16]

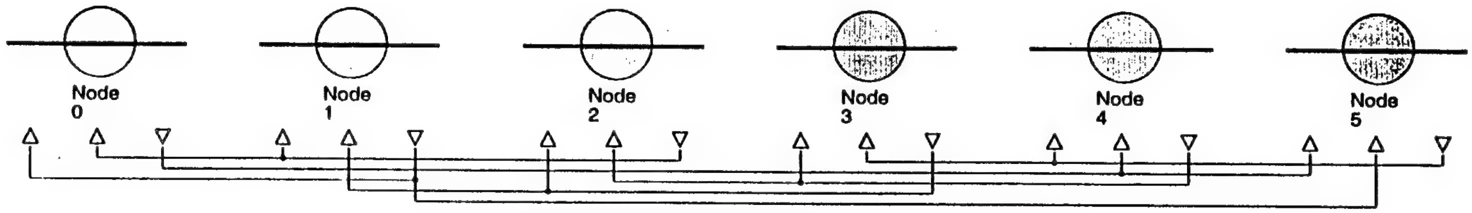


Trigger on X Count [17-Feb-1997 12:03:36]



This module simply counts the inputs arriving at the input clock and generates an output at X # of clocks as set by the argument "Trigger after X counts".

Six Node Ring [17-Feb-1997 12:04:13]



This is a six node token ring

Data Link Layer Layer Arguments

- ⇑P Token Hold Factor
- ⇑P LPCoefficient
- ⇑P LLC Queue Length
- ⇑P OTDelay
- ⇑P Average Pulse Width
- ⇑P Percentage of Clock Skew

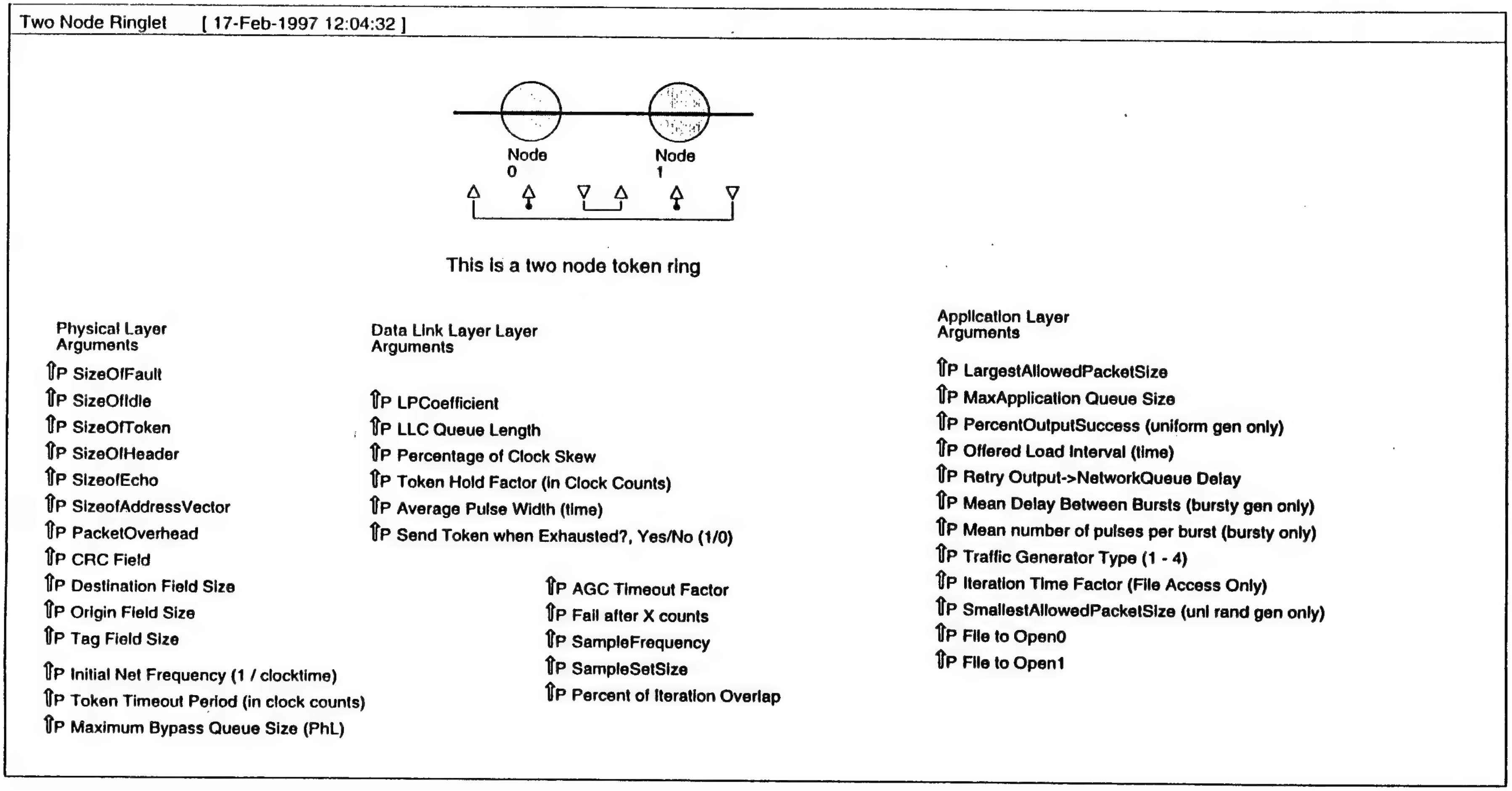
- ⇑P AGC Timeout Factor
- ⇑P Fail after X counts
- ⇑P SampleTime
- ⇑P SampleSetSize
- ⇑P #ofIterations

Physical Layer Arguments

- ⇑P SizeOfFault
- ⇑P SizeOfIdle
- ⇑P SizeOfToken
- ⇑P SizeOfHeader
- ⇑P SizeofEcho
- ⇑P SizeofDataHeader
- ⇑P SizeofAddressVector
- ⇑P PacketOverhead
- ⇑P CRC Field
- ⇑P Destination Field Size
- ⇑P Origin Field Size
- ⇑P Tag Field Size
- ⇑P # Token wait counts
- ⇑P Initial Net Frequency

Application Layer Arguments

- ⇑P Mean Delay Between Bursts
- ⇑P Mean number of pulses per burst
- ⇑P Traffic Generator Type
- ⇑P Iteration Time Factor
- ⇑P File to Open0
- ⇑P File to Open1
- ⇑P File to Open2
- ⇑P File to Open3
- ⇑P File to Open4
- ⇑P File to Open5
- ⇑P PercentOutputSuccess
- ⇑P LargestAllowedPacketSize
- ⇑P Inter-Pulse Time
- ⇑P Retry Output Delay
- ⇑P MaxApp Queue Size
- ⇑P SmallestAllowedPacketSize



Two Node Ringlet [17-Feb-1997 12:04:32]

This is a two node token ring

Physical Layer
Arguments

⇑P SizeOfFault
⇑P SizeOfIdle
⇑P SizeOfToken
⇑P SizeOfHeader
⇑P SizeofEcho
⇑P SizeofAddressVector
⇑P PacketOverhead
⇑P CRC Field
⇑P Destination Field Size
⇑P Origin Field Size
⇑P Tag Field Size
⇑P Initial Net Frequency (1 / clocktime)
⇑P Token Timeout Period (in clock counts)
⇑P Maximum Bypass Queue Size (PhL)

Data Link Layer Layer
Arguments

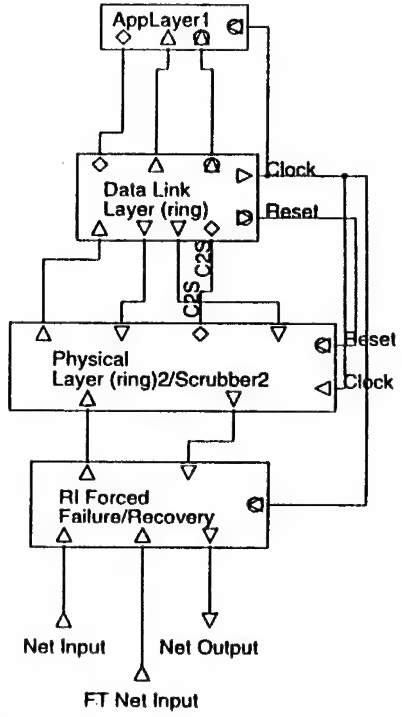
⇑P LPCoefficient
⇑P LLC Queue Length
⇑P Percentage of Clock Skew
⇑P Token Hold Factor (in Clock Counts)
⇑P Average Pulse Width (time)
⇑P Send Token when Exhausted?, Yes/No (1/0)

⇑P AGC Timeout Factor
⇑P Fail after X counts
⇑P SampleFrequency
⇑P SampleSetSize
⇑P Percent of Iteration Overlap

Application Layer
Arguments

⇑P LargestAllowedPacketSize
⇑P MaxApplication Queue Size
⇑P PercentOutputSuccess (uniform gen only)
⇑P Offered Load Interval (time)
⇑P Retry Output->NetworkQueue Delay
⇑P Mean Delay Between Bursts (bursty gen only)
⇑P Mean number of pulses per burst (bursty only)
⇑P Traffic Generator Type (1 - 4)
⇑P Iteration Time Factor (File Access Only)
⇑P SmallestAllowedPacketSize (uni rand gen only)
⇑P File to Open0
⇑P File to Open1

Ring Node [17-Feb-1997 12:04:45]



This is the Token Ring Node.

App Layer Arguments

⇑P Number of Nodes
⇑P LargestAllowedPacketSize
⇑P File to Open
⇑P MaxApplication Queue Size
⇑P PercentOutputSuccess (uniform gen only)
⇑P Offered Load Interval

⇑P Retry Output->NetworkQueue Delay
⇑P Mean Delay Between Bursts (bursty gen only)
⇑P Mean number of pulses per burst (bursty only)
⇑P Traffic Generator Type (1 - 4)
⇑P Iteration Time Factor (File Access Only)
⇑P SmallestAllowedPacketSize (uni rand gen only)

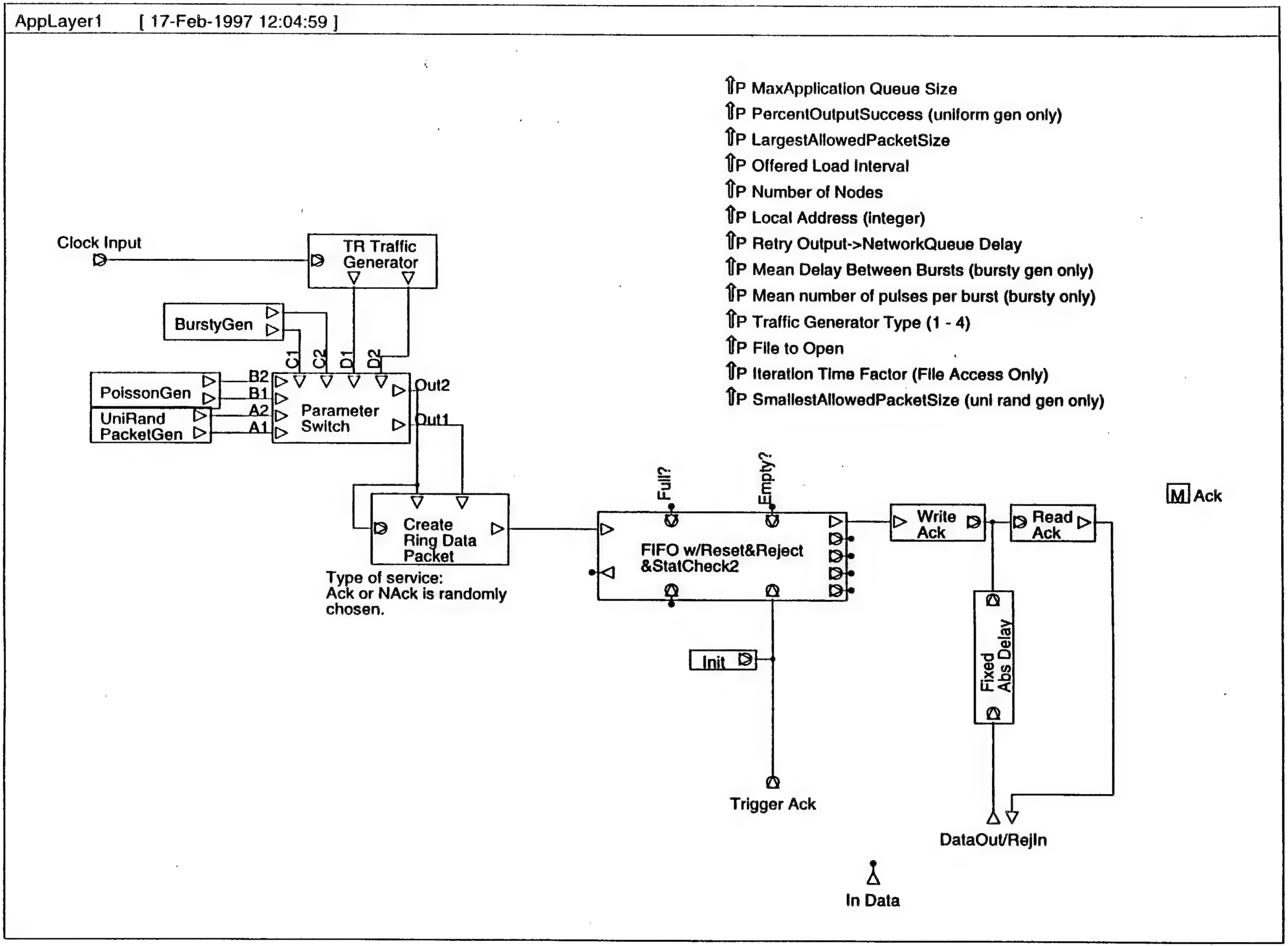
Physical Layer
Arguments

⇑P SizeOfFault
⇑P SizeOfIdle
⇑P SizeOfToken
⇑P SizeOfHeader
⇑P SizeofEcho
⇑P SizeofAddressVector
⇑P PacketOverhead
⇑P Initial Net Frequency (1 / clocktime)
⇑P Token Timeout Period (in clock counts)
⇑P Maximum Bypass Queue Size (PhL)

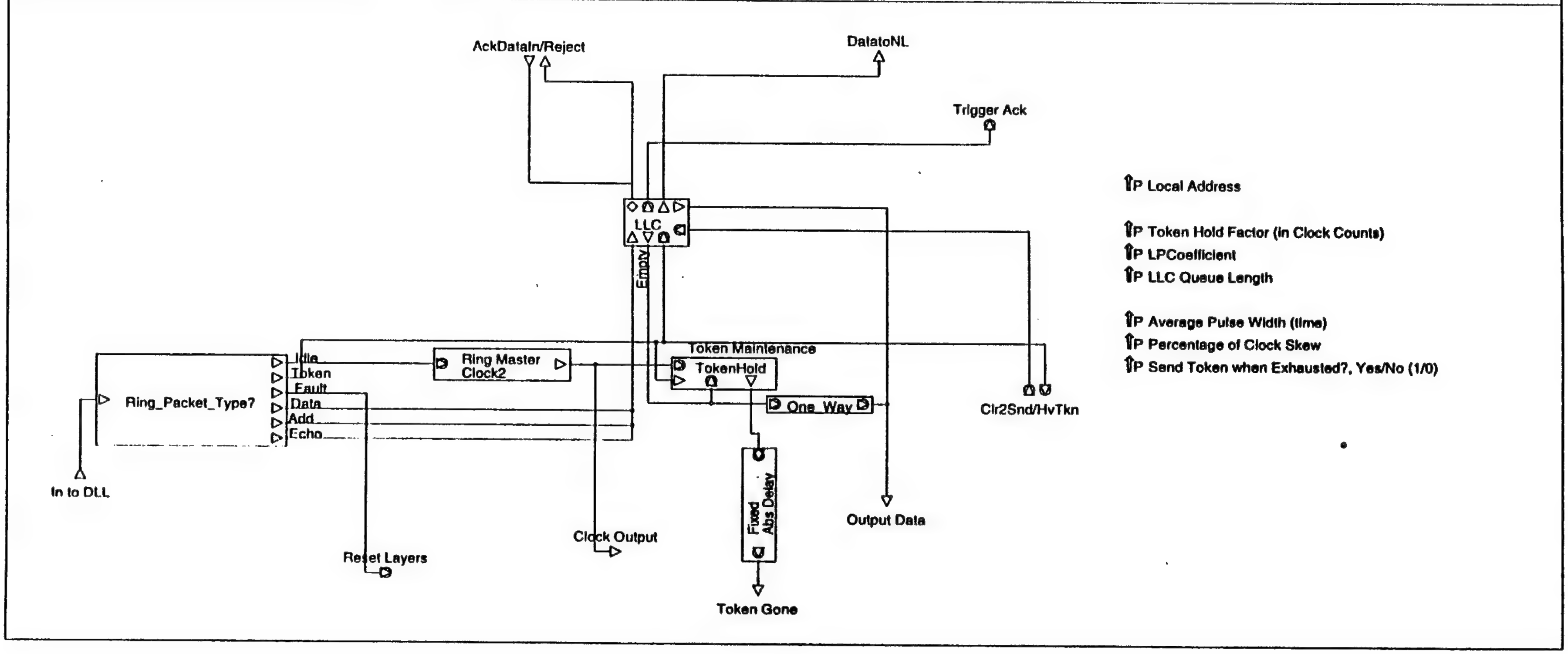
DLL Arguments

⇑P Token Hold Factor (in Clock Counts)
⇑P LPCoefficient
⇑P LLC Queue Length
⇑P Average Pulse Width (time)
⇑P Percentage of Clock Skew
⇑P Send Token when Exhausted?, Yes/No (1/0)

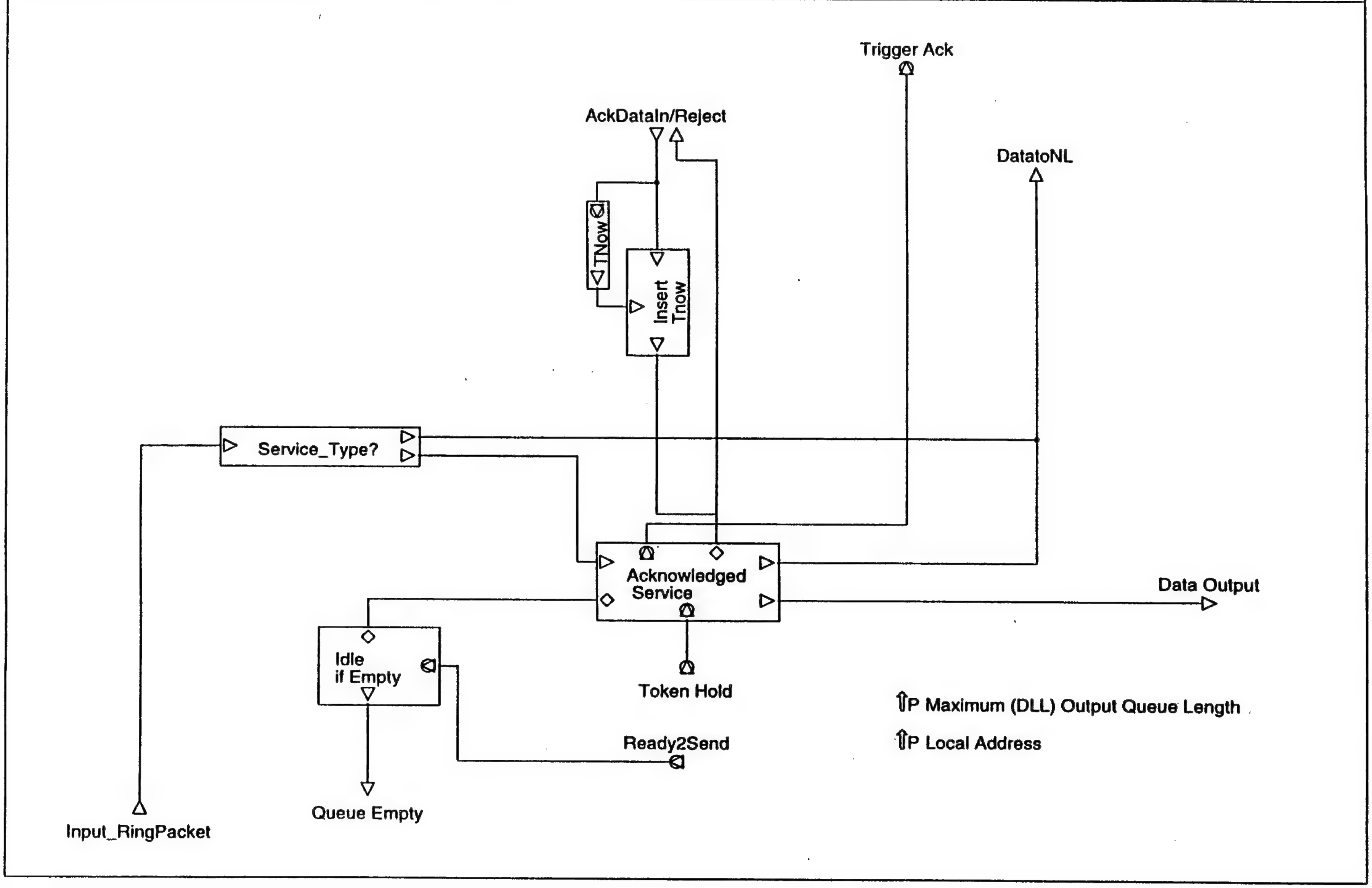
⇑P AGC Timeout Factor
⇑P Fail after X counts
⇑P Local Address


AppLayer1 [17-Feb-1997 12:04:59]
⇑P MaxApplication Queue Size
⇑P PercentOutputSuccess (uniform gen only)
⇑P LargestAllowedPacketSize
⇑P Offered Load Interval
⇑P Number of Nodes
⇑P Local Address (integer)
⇑P Retry Output->NetworkQueue Delay
⇑P Mean Delay Between Bursts (bursty gen only)
⇑P Mean number of pulses per burst (bursty only)
⇑P Traffic Generator Type (1 - 4)
⇑P File to Open
⇑P Iteration Time Factor (File Access Only)
⇑P SmallestAllowedPacketSize (uni rand gen only)
Clock Input
TR Traffic Generator
BurstyGen
PoissonGen
UniRand PacketGen
B2
B1
A2
A1
C1
C2
D1
D2
Parameter Switch
Out2
Out1
Create Ring Data Packet
Type of service:
Ack or NAck is randomly chosen.
Full?
Empty?
FIFO w/Reset&Reject &StatCheck2
Init
Trigger Ack
Write Ack
Read Ack
Fixed Abs Delay
M Ack
DataOut/RejIn
In Data

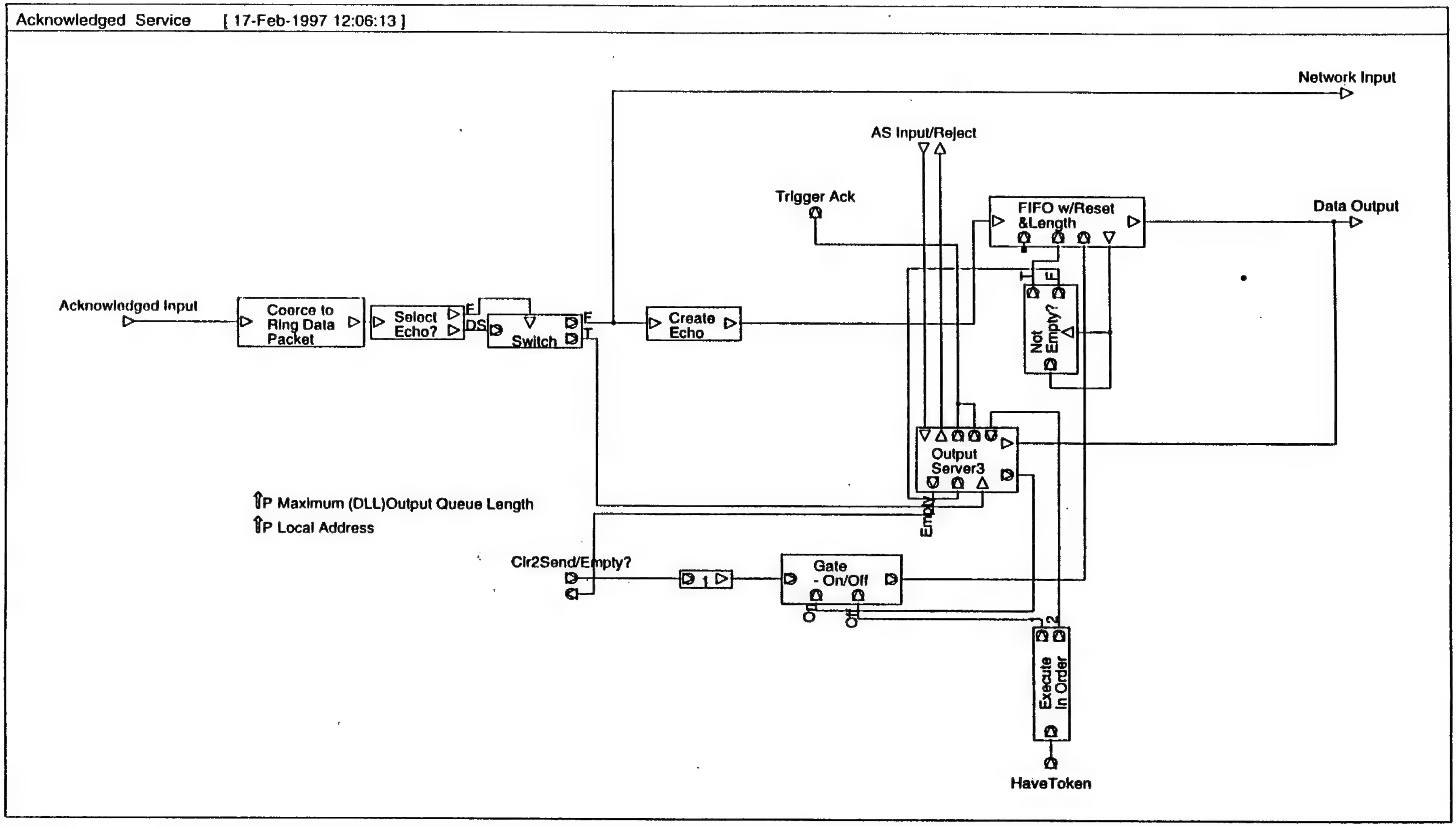
Data Link Layer (ring) [17-Feb-1997 12:05:39]



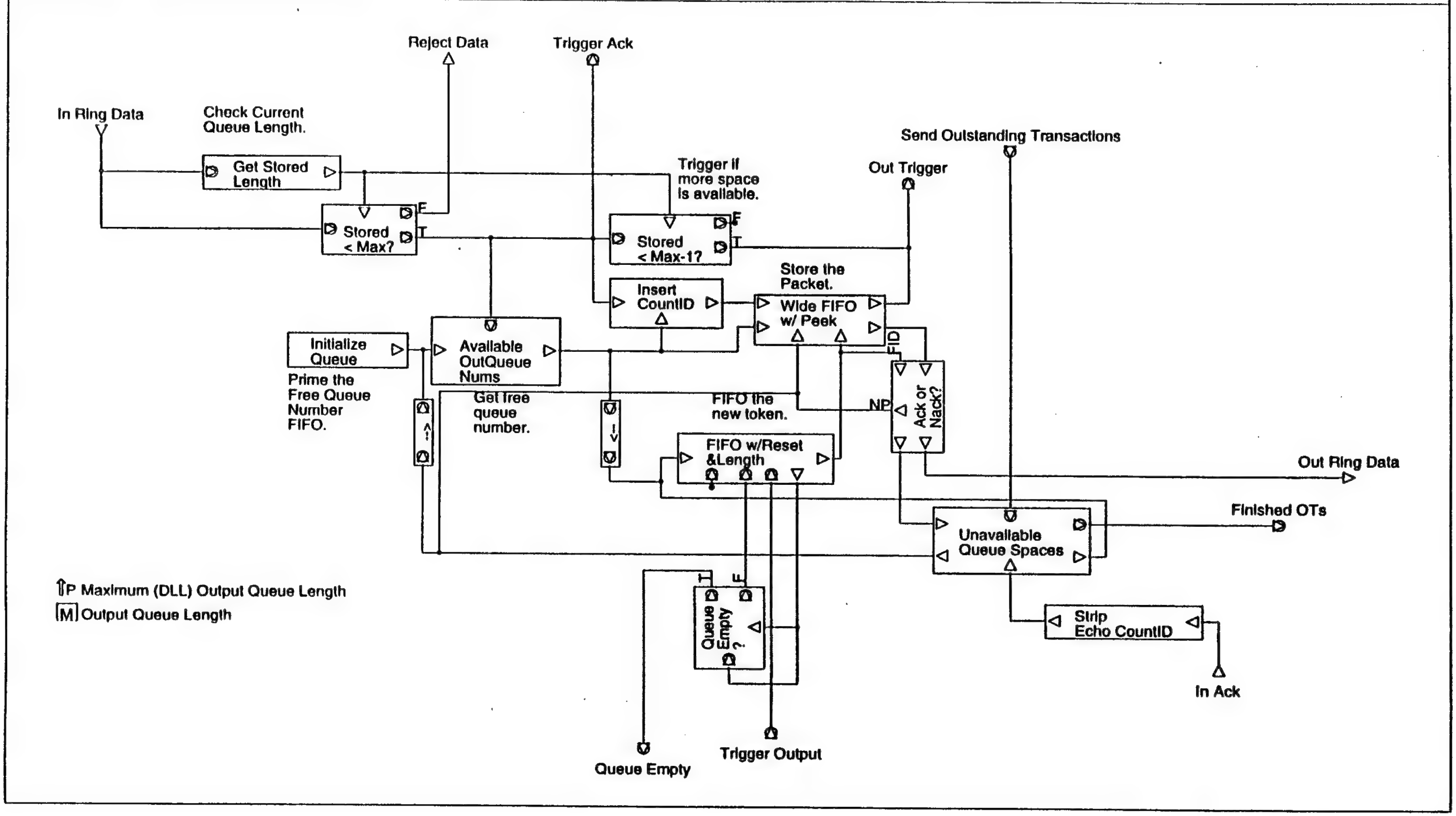
LLC [17-Feb-1997 12:05:52]



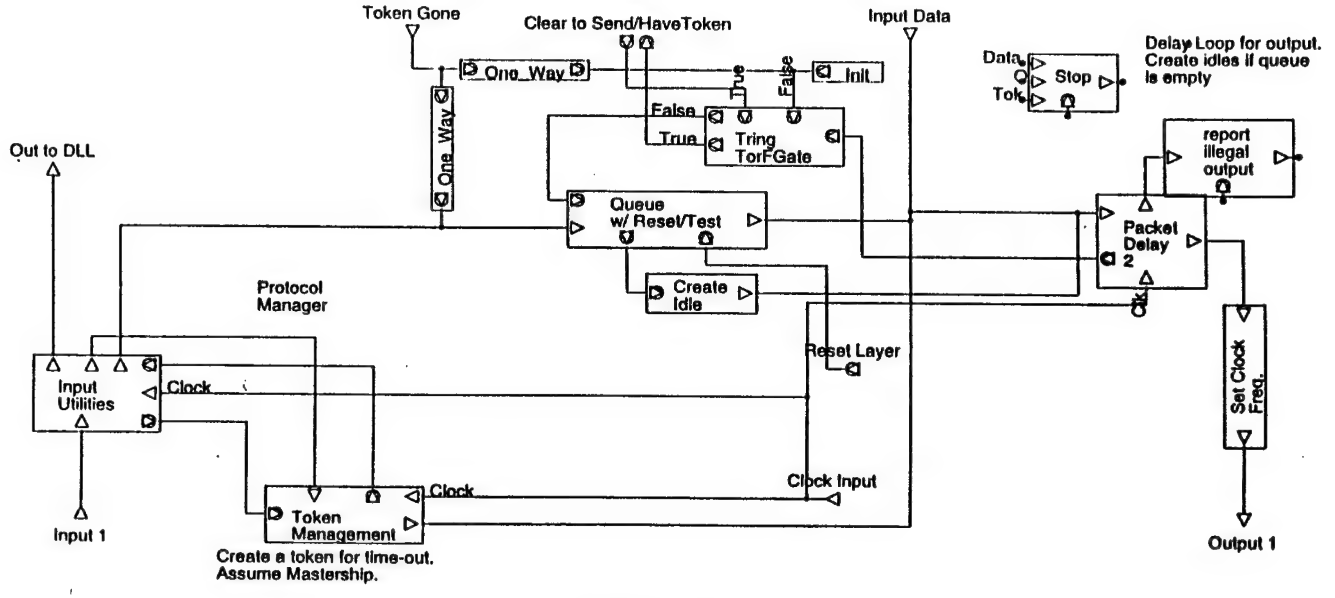
Acknowledged Service [17-Feb-1997 12:06:13]



Output Server3 [17-Feb-1997 12:06:23]



Physical Layer (ring)2/Scrubber2 [17-Feb-1997 12:06:56]

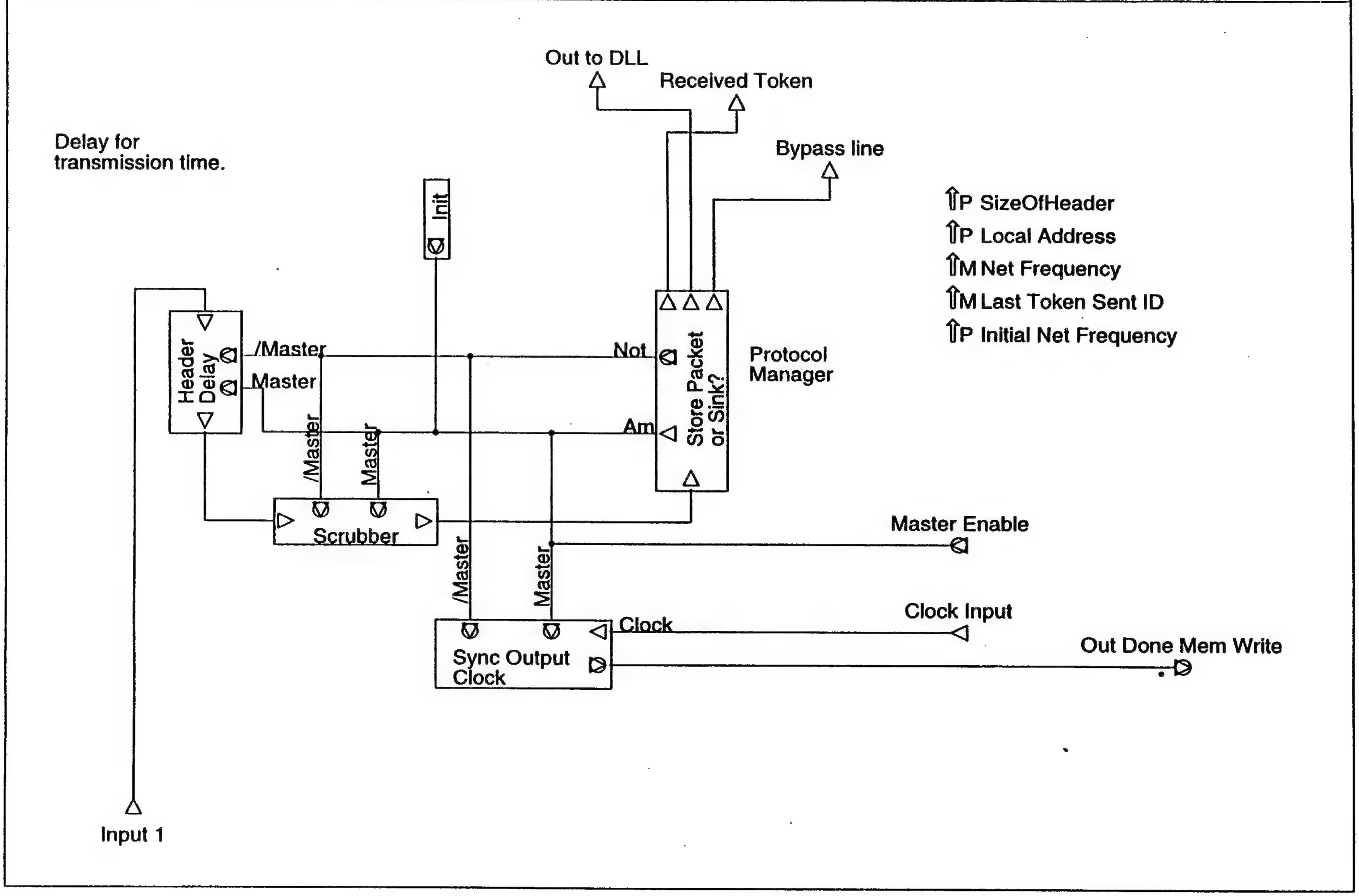


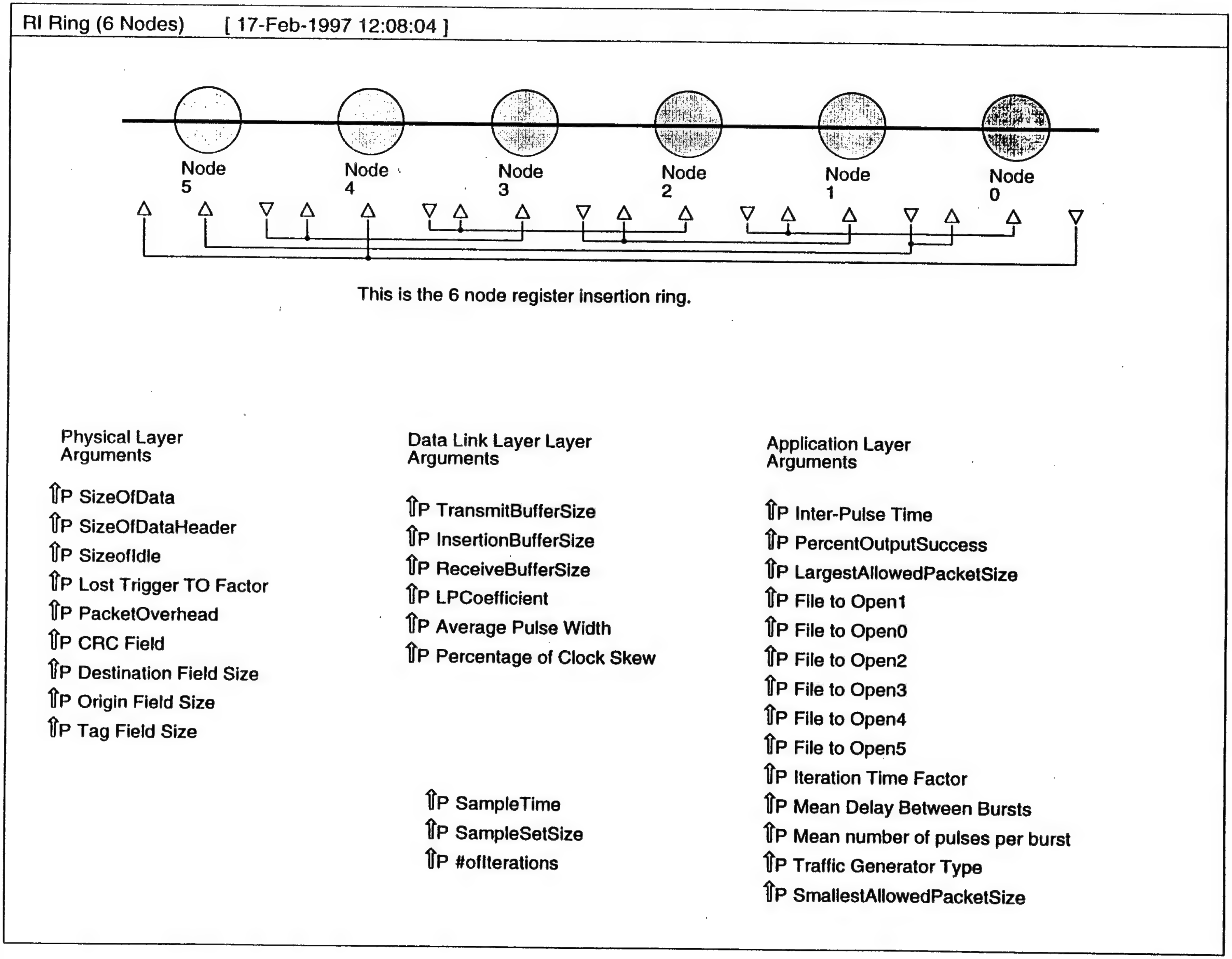
M Net Frequency
M Last Token Sent ID

⇧P SizeofAddressVector
⇧P SizeofEcho
⇧P SizeOfFault
⇧P SizeOfIdle
⇧P SizeOfToken
⇧P SizeOfHeader
⇧P PacketOverhead

⇧P Local Address
⇧P Token Timeout Period (in clock counts)
⇧P Maximum Bypass Queue Size (PhL)
⇧P Initial Net Frequency (1 / clocktime)

Input Utilities [17-Feb-1997 12:07:21]





RI Ring (6 Nodes) [17-Feb-1997 12:08:04]

This is the 6 node register insertion ring.

Physical Layer Arguments

⇧P SizeOfData
⇧P SizeOfDataHeader
⇧P SizeofIdle
⇧P Lost Trigger TO Factor
⇧P PacketOverhead
⇧P CRC Field
⇧P Destination Field Size
⇧P Origin Field Size
⇧P Tag Field Size

Data Link Layer Layer Arguments

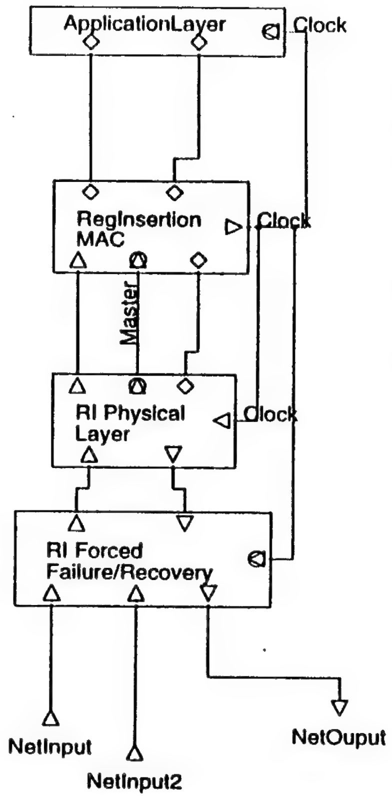
⇧P TransmitBufferSize
⇧P InsertionBufferSize
⇧P ReceiveBufferSize
⇧P LPCoefficient
⇧P Average Pulse Width
⇧P Percentage of Clock Skew

⇧P SampleTime
⇧P SampleSetSize
⇧P #ofIterations

Application Layer Arguments

⇧P Inter-Pulse Time
⇧P PercentOutputSuccess
⇧P LargestAllowedPacketSize
⇧P File to Open1
⇧P File to Open0
⇧P File to Open2
⇧P File to Open3
⇧P File to Open4
⇧P File to Open5
⇧P Iteration Time Factor
⇧P Mean Delay Between Bursts
⇧P Mean number of pulses per burst
⇧P Traffic Generator Type
⇧P SmallestAllowedPacketSize

RegInsertion Node [17-Feb-1997 12:08:24]



Physical Layer Arguments

⇧P Master?
⇧P SizeOfData
⇧P SizeOfDataHeader
⇧P Lost Trigger TO Factor
⇧P Sizeofldle

⇧P AGC Timeout Factor
⇧P Fail after X counts
⇧P Local_address
⇧P PacketOverhead

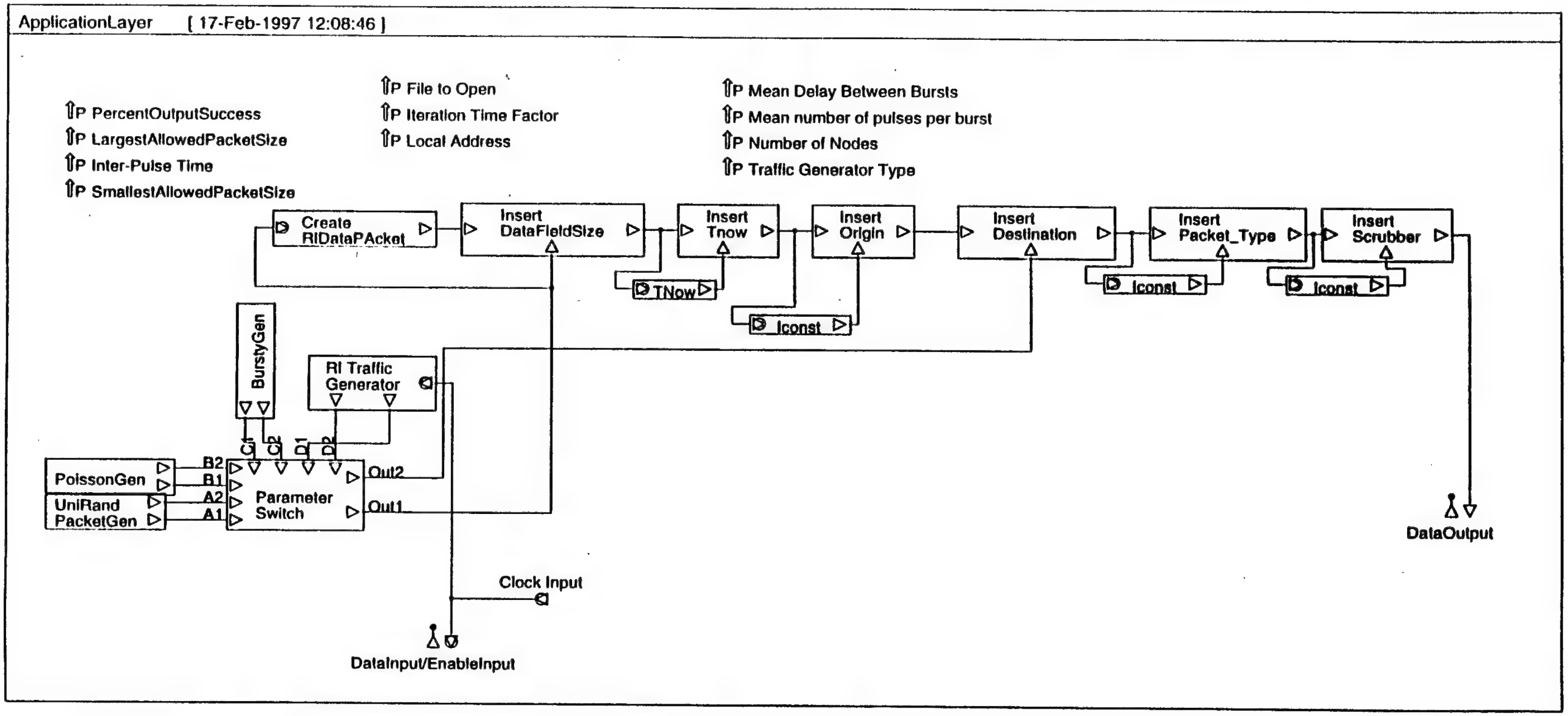
Data Link Layer Layer Arguments

⇧P TransmitBufferSize
⇧P InsertionBufferSize
⇧P ReceiveBufferSize
⇧P LPCoefficient
⇧P Average Pulse Width
⇧P Percentage of Clock Skew

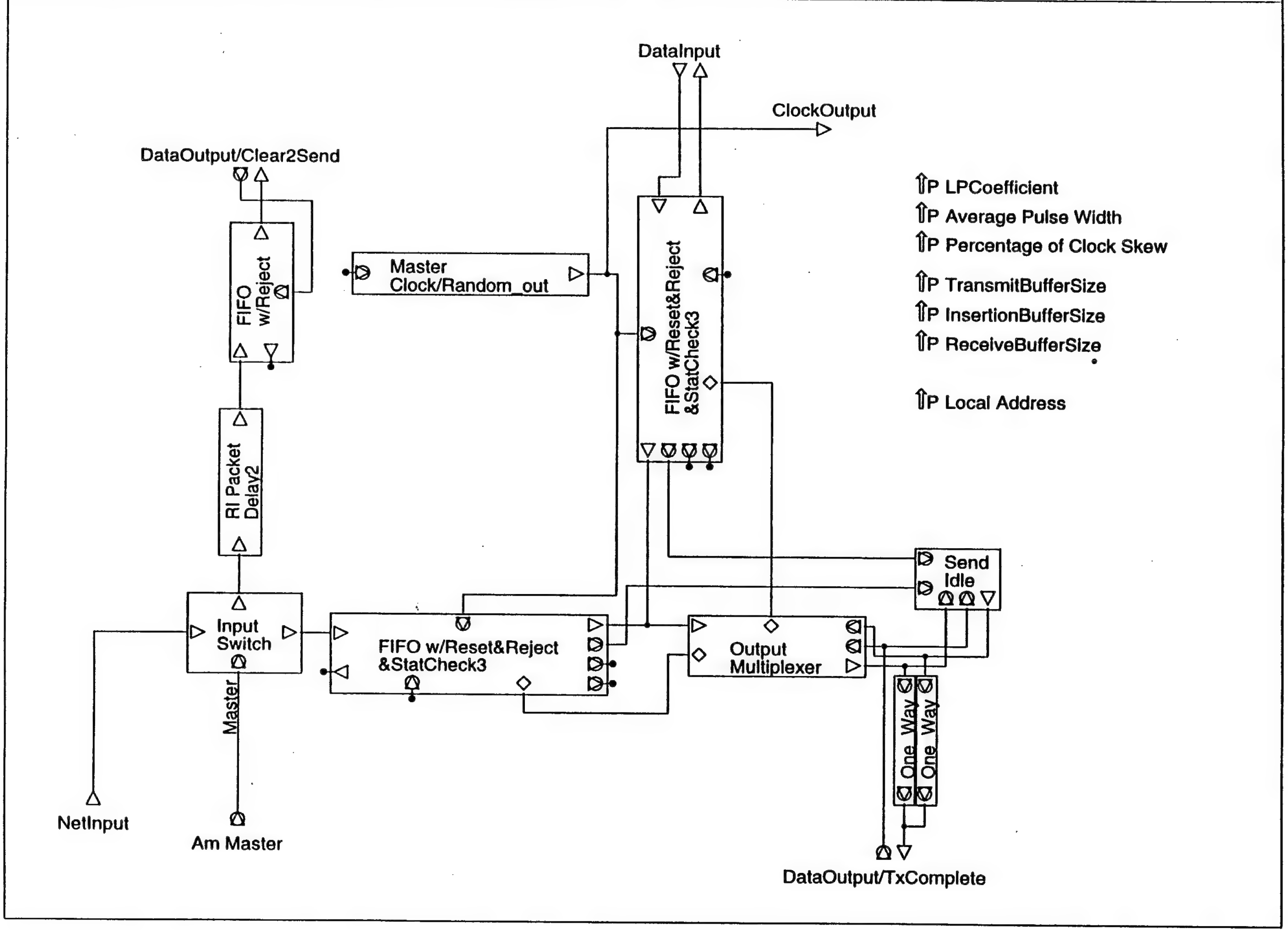
Application Layer Arguments

⇧P Inter-Pulse Time
⇧P PercentOutputSuccess
⇧P LargestAllowedPacketSize
⇧P File to Open
⇧P Iteration Time Factor
⇧P Mean Delay Between Bursts
⇧P Mean number of pulses per burst
⇧P Number of Nodes
⇧P Traffic Generator Type
⇧P SmallestAllowedPacketSize

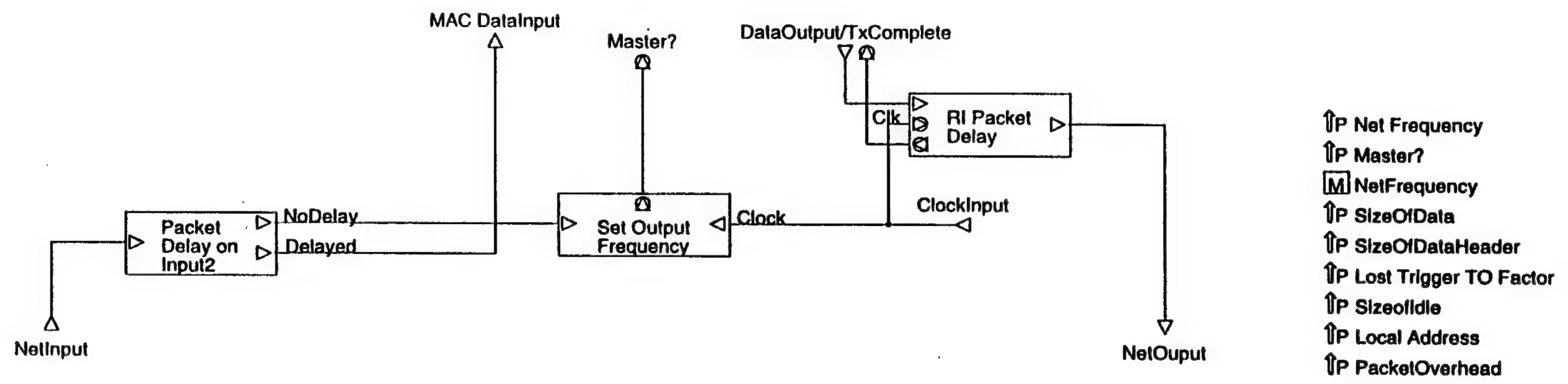
This is the register insertion node.


ApplicationLayer [17-Feb-1997 12:08:46]
P PercentOutputSuccess
P LargestAllowedPacketSize
P Inter-Pulse Time
P SmallestAllowedPacketSize
P File to Open
P Iteration Time Factor
P Local Address
P Mean Delay Between Bursts
P Mean number of pulses per burst
P Number of Nodes
P Traffic Generator Type
Create RlDataPAcket
Insert DataFieldSize
Insert Tnow
TNow
Insert Origin
Iconst
Insert Destination
Insert Packet_Type
Iconst
Insert Scrubber
Iconst
BurstyGen
Rl Traffic Generator
PoissonGen
UniRand PacketGen
B2
B1
A2
A1
C1
C2
D1
D2
Parameter Switch
Out2
Out1
Clock Input
DataInput/EnableInput
DataOutput

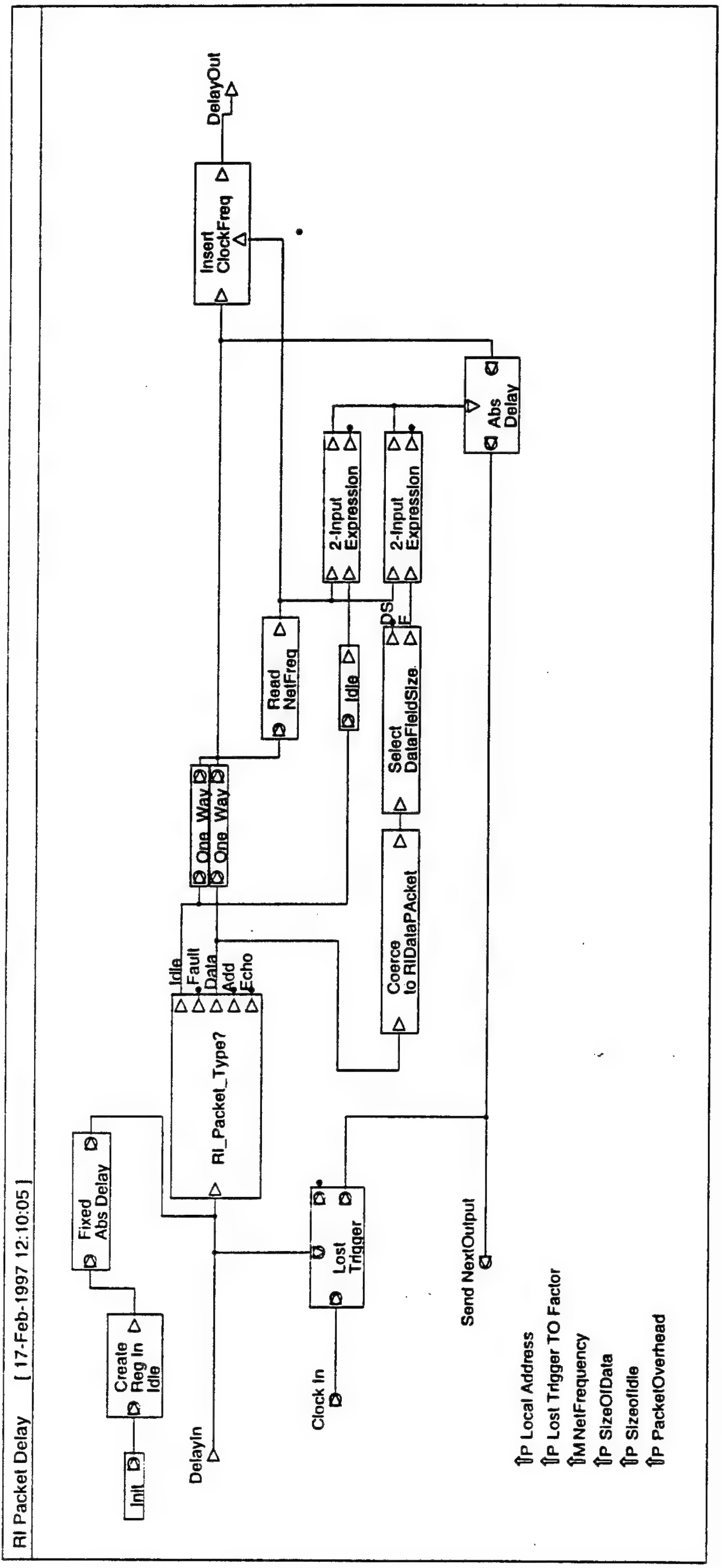
RI_MAC [17-Feb-1997 12:09:16]



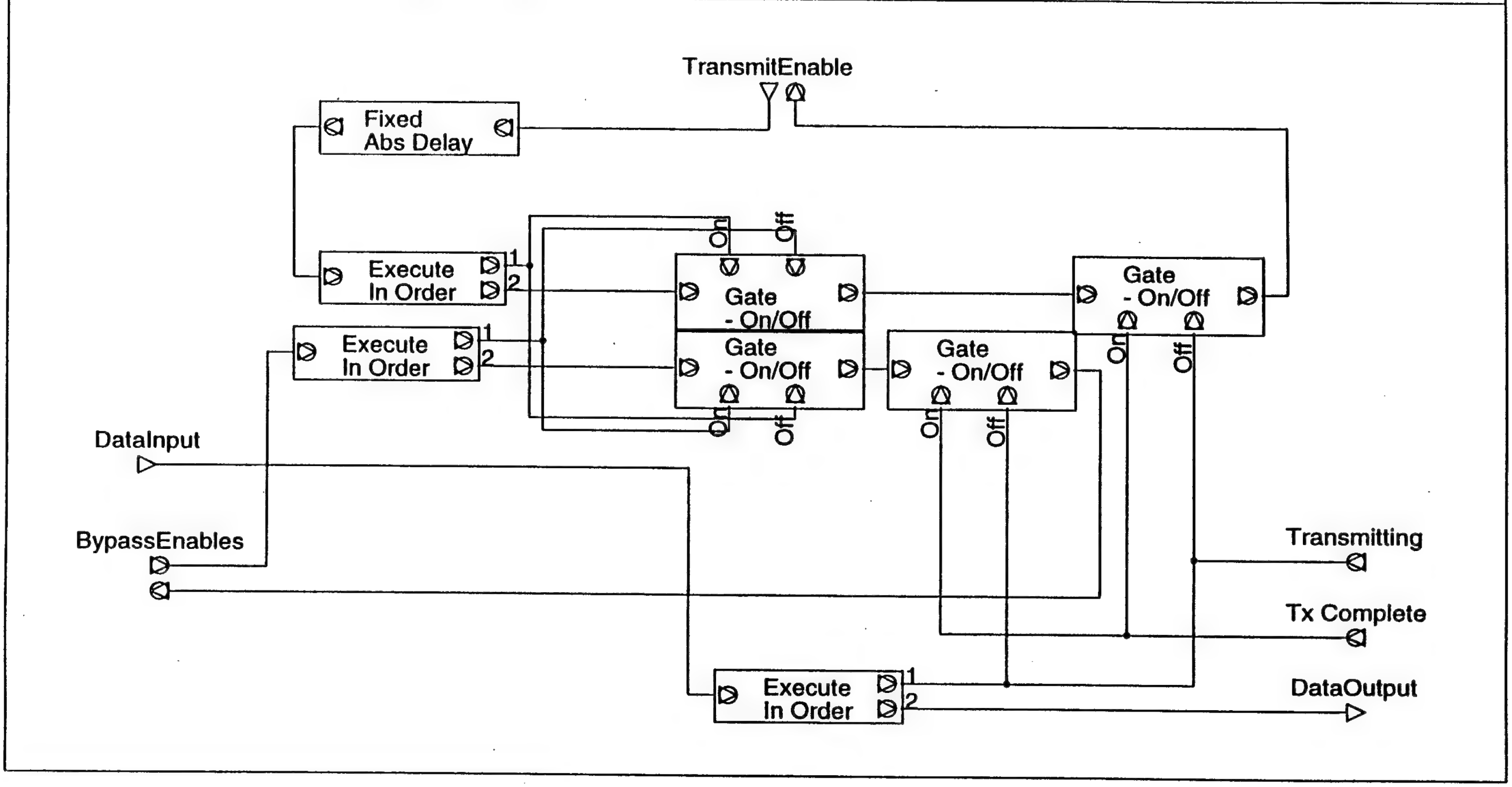
RI Physical Layer [17-Feb-1997 12:09:54]



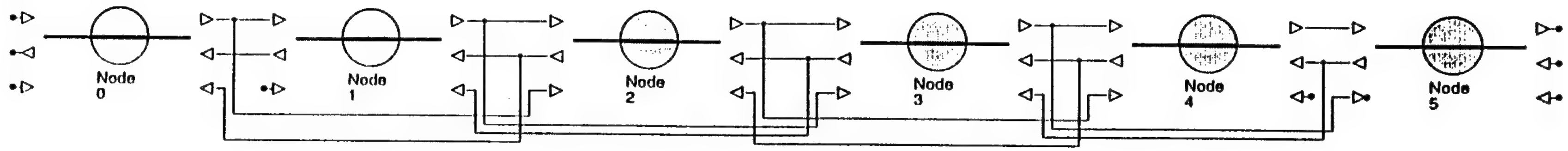
RI Packet Delay [17-Feb-1997 12:10:05]



Output Multiplexer [17-Feb-1997 12:10:23]



Six Node Bi-Array [17-Feb-1997 12:10:52]



This is a six node bi-directional array.

Physical Layer Arguments

- ⇧P Master Timeout Factor
- ⇧P Master?
- ⇧P SizeofIdle
- ⇧P SizeOfData
- ⇧P SizeOfDataHeader
- ⇧P Lost Trigger TO Factor
- ⇧P PacketOverhead
- ⇧P CRC Field
- ⇧P Destination Field Size
- ⇧P Origin Field Size
- ⇧P Tag Field Size

Data Link Layer Layer Arguments

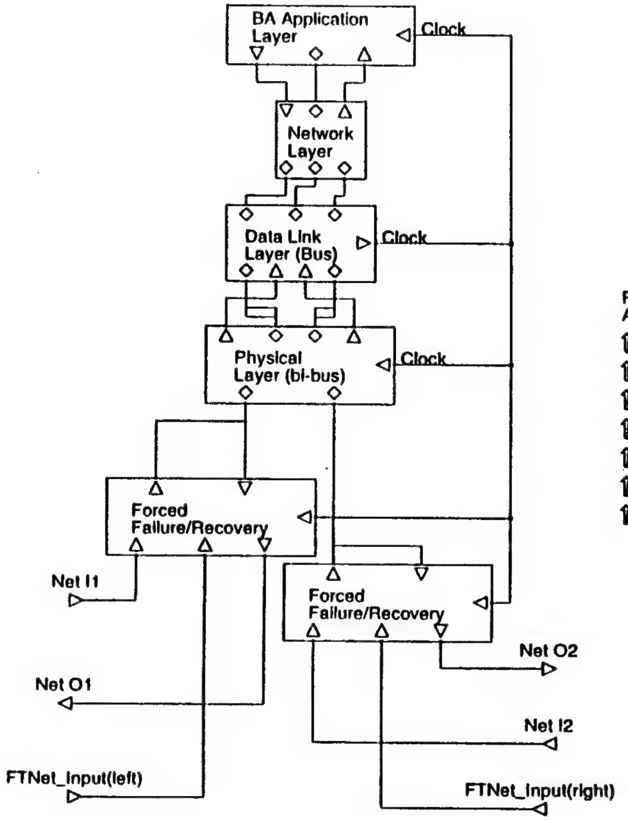
- ⇧P TransmitBufferSize
- ⇧P InsertionBufferSize
- ⇧P ReceiveBufferSize
- ⇧P LPCoefficient
- ⇧P Average Pulse Width
- ⇧P Percentage of Clock Skew

Application Layer Arguments

- ⇧P PercentOutputSuccess
- ⇧P LargestAllowedPacketSize
- ⇧P Inter-Pulse Time
- ⇧P Mean Delay Between Bursts
- ⇧P Mean number of pulses per burst
- ⇧P File to Open0
- ⇧P File to Open1
- ⇧P File to Open2
- ⇧P File to Open3
- ⇧P File to Open4
- ⇧P File to Open5
- ⇧P Iteration Time Factor
- ⇧P Traffic Generator Type
- ⇧P SmallestAllowedPacketSize

- ⇧P AGC Timeout Factor
- ⇧P SampleTime
- ⇧P SampleSetSize
- ⇧P #ofIterations

Bi-Bus Node   [17-Feb-1997 12:11:06]



This is the bi-directional node.

Physical Layer Arguments

⇧P Master Timeout Factor
⇧P Master?
⇧P SizeofIdle
⇧P SizeOfData
⇧P SizeOfDataHeader
⇧P Lost Trigger TO Factor
⇧P PacketOverhead

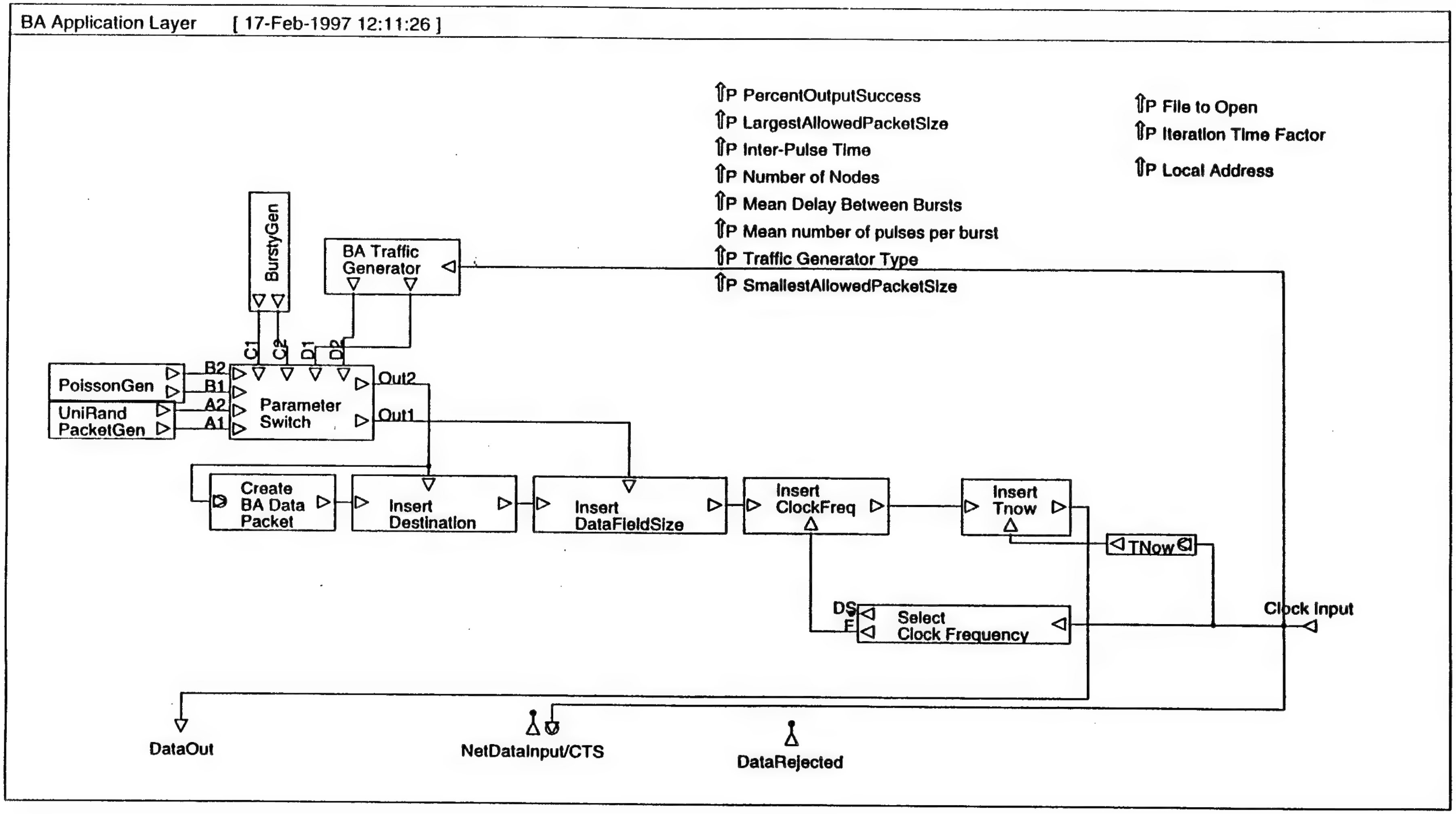
Data Link Layer Layer Arguments

⇧P TransmitBufferSize
⇧P InsertionBufferSize
⇧P ReceiveBufferSize
⇧P LPCoefficient
⇧P Average Pulse Width
⇧P Percentage of Clock Skew

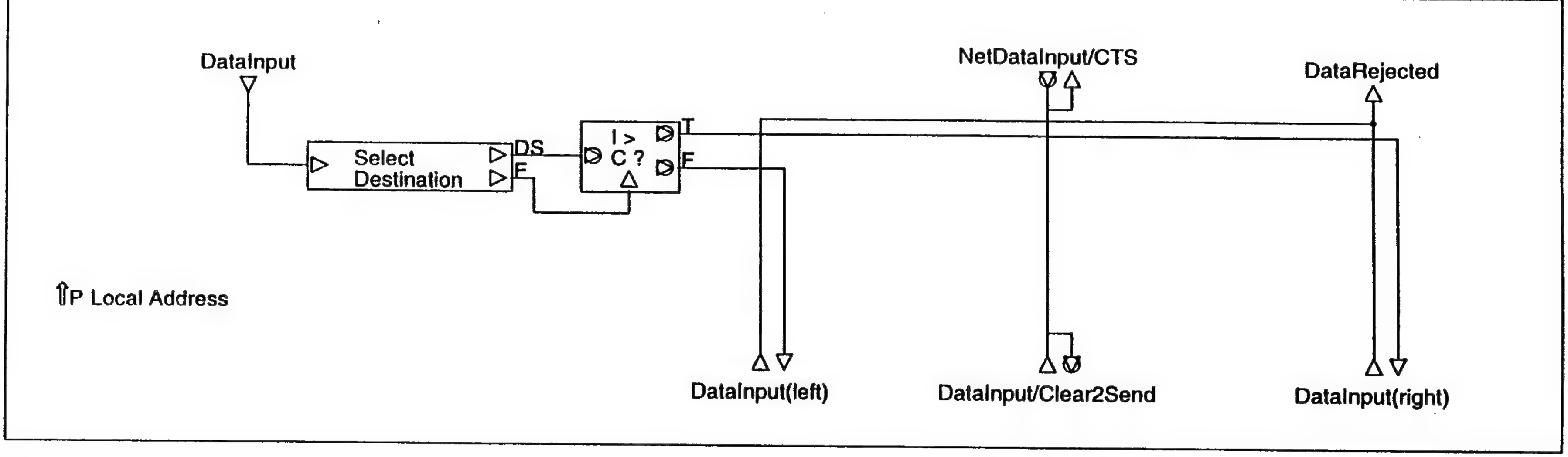
⇧P AGC Timeout Factor
⇧P Fail after X counts
⇧P Local_address

Application Layer Arguments

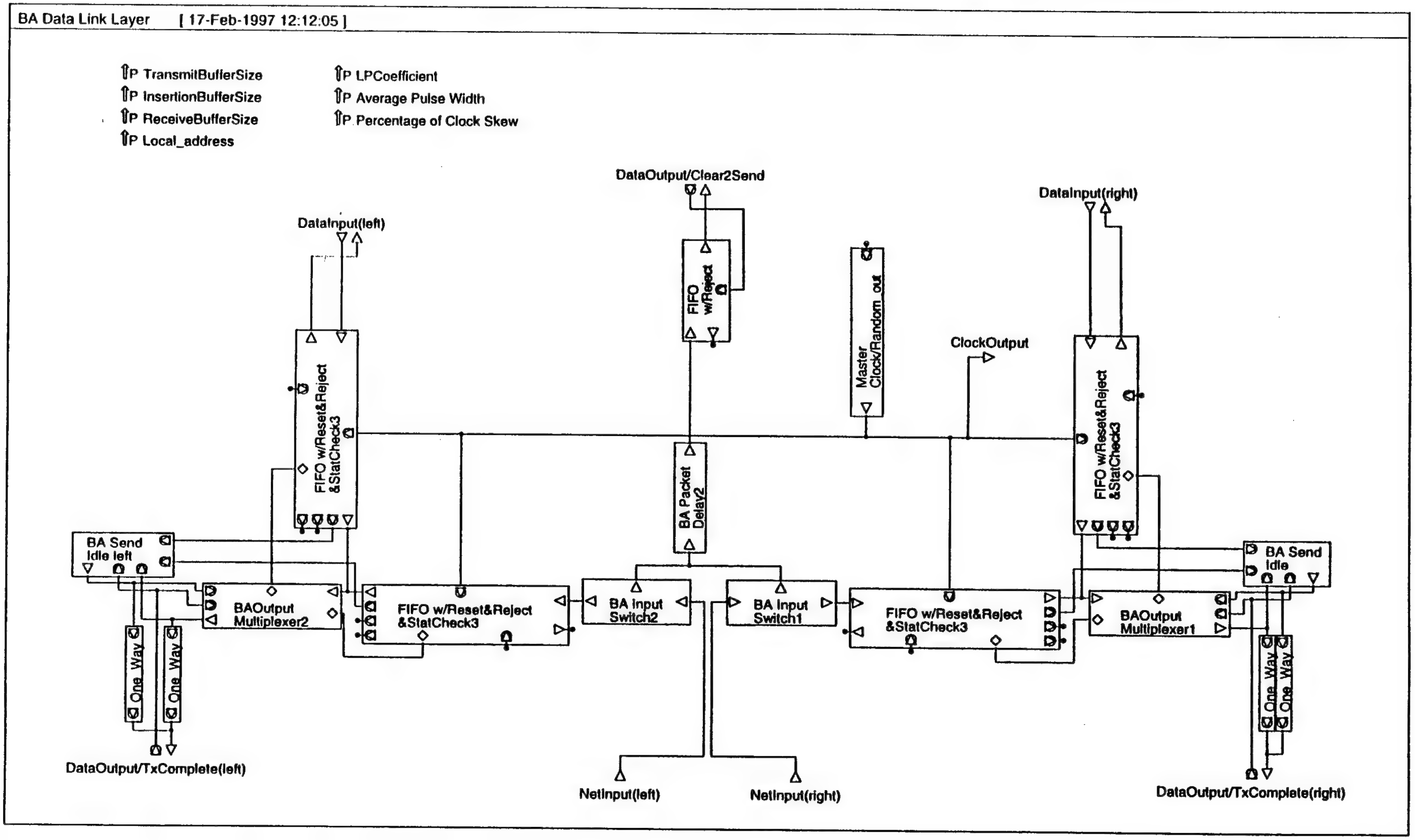
⇧P File to Open
⇧P Iteration Time Factor
⇧P PercentOutputSuccess
⇧P LargestAllowedPacketSize
⇧P Inter-Pulse Time
⇧P Number of Nodes
⇧P Mean Delay Between Bursts
⇧P Mean number of pulses per burst
⇧P Traffic Generator Type
⇧P SmallestAllowedPacketSize


BA Application Layer [17-Feb-1997 12:11:26]
⇧P PercentOutputSuccess
⇧P LargestAllowedPacketSize
⇧P Inter-Pulse Time
⇧P Number of Nodes
⇧P Mean Delay Between Bursts
⇧P Mean number of pulses per burst
⇧P Traffic Generator Type
⇧P SmallestAllowedPacketSize
⇧P File to Open
⇧P Iteration Time Factor
⇧P Local Address
BurstyGen
BA Traffic Generator
C1
C2
D1
D2
PoissonGen
UniRand PacketGen
B2
B1
A2
A1
Parameter Switch
Out2
Out1
Create BA Data Packet
Insert Destination
Insert DataFieldSize
Insert ClockFreq
Insert Tnow
TNow
DS
F
Select Clock Frequency
Clock Input
DataOut
NetDataInput/CTS
DataRejected

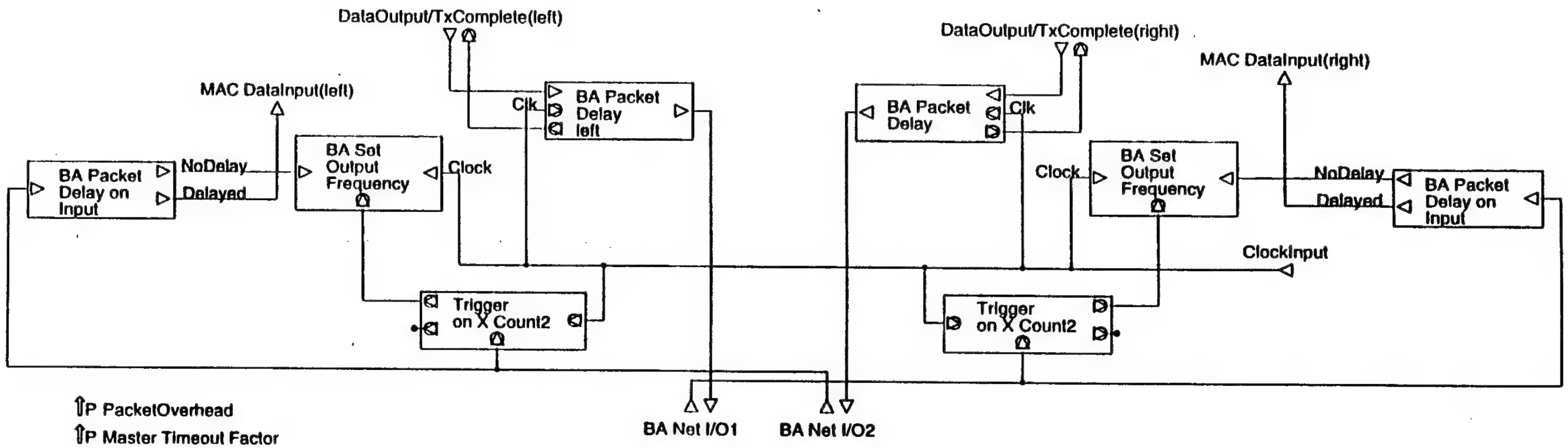
Network Layer [17-Feb-1997 12:11:48]



BA Data Link Layer [17-Feb-1997 12:12:05]



BA Physical Layer [17-Feb-1997 12:12:20]

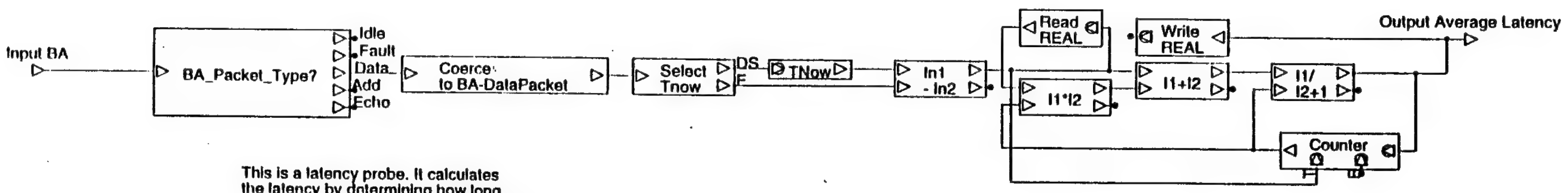


⇑P PacketOverhead
⇑P Master Timeout Factor
⇑P Net Frequency
⇑P Master?
⇑P SizeofIdle
⇑P SizeOfData
⇑P SizeOfDataHeader
⇑P Lost Trigger TO Factor

⇑P Local Address

M NetFrequency(right)
M NetFrequency(left)

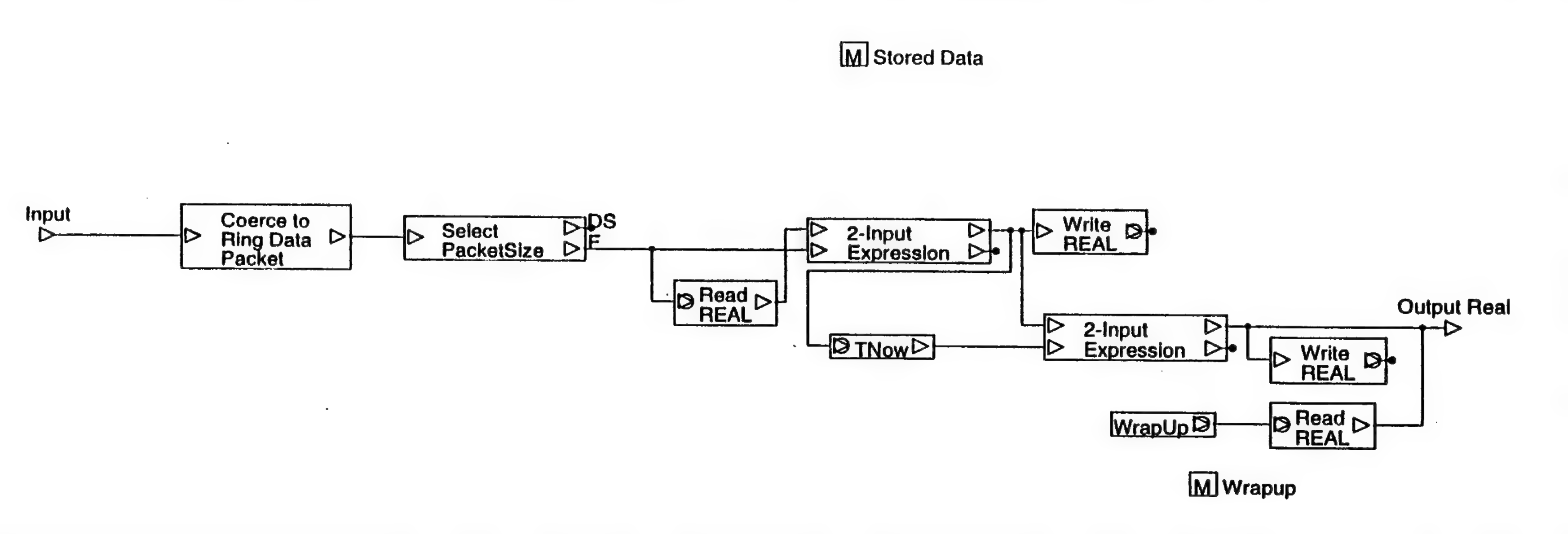
BA Average Latency Probe [17-Feb-1997 11:57:09]



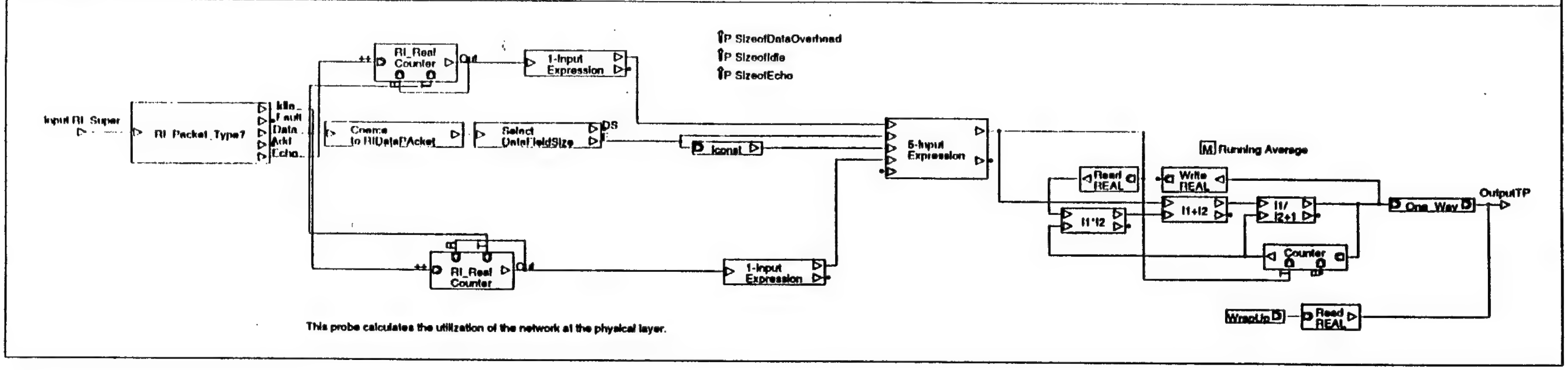
This is a latency probe. It calculates the latency by determining how long it took to propagate down the network by subtracting the current time from the time the data was sent into the network layer.

M Running Average

TR Application Specific TP [17-Feb-1997 11:57:49]

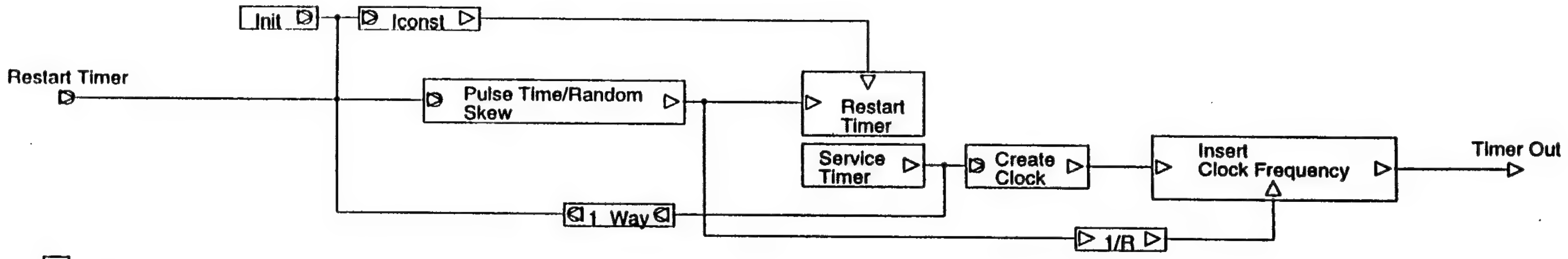


RA RI_Utilization_Probe   [17-Feb-1997 11:58:31]



This probe calculates the utilization of the network at the physical layer.

Master Clock/Random_out [17-Feb-1997 11:59:29]



E Master/Slave

⇧P LPCoefficient

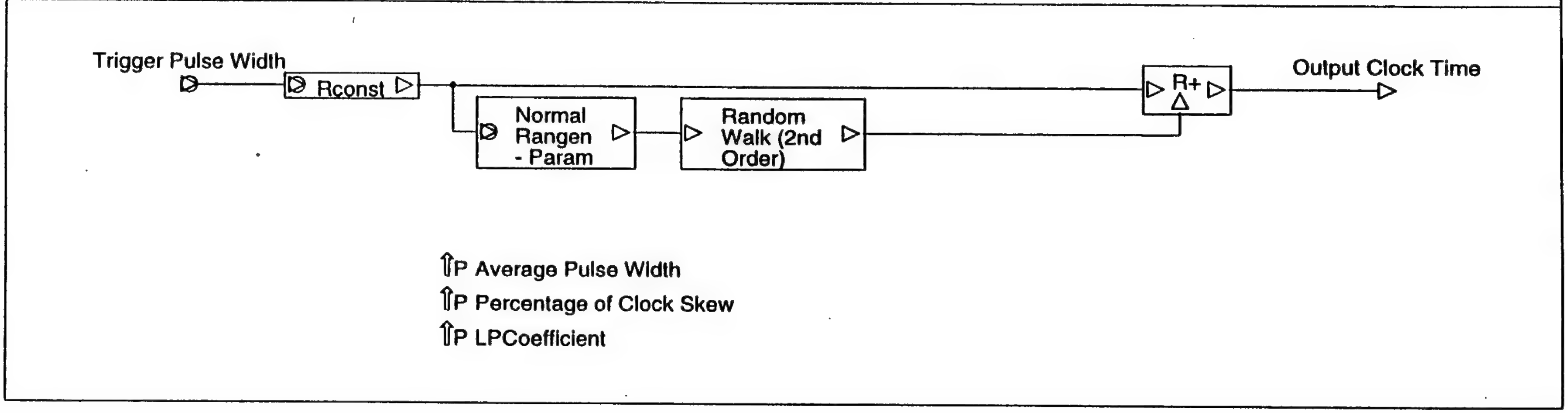
⇧P Average Pulse Width

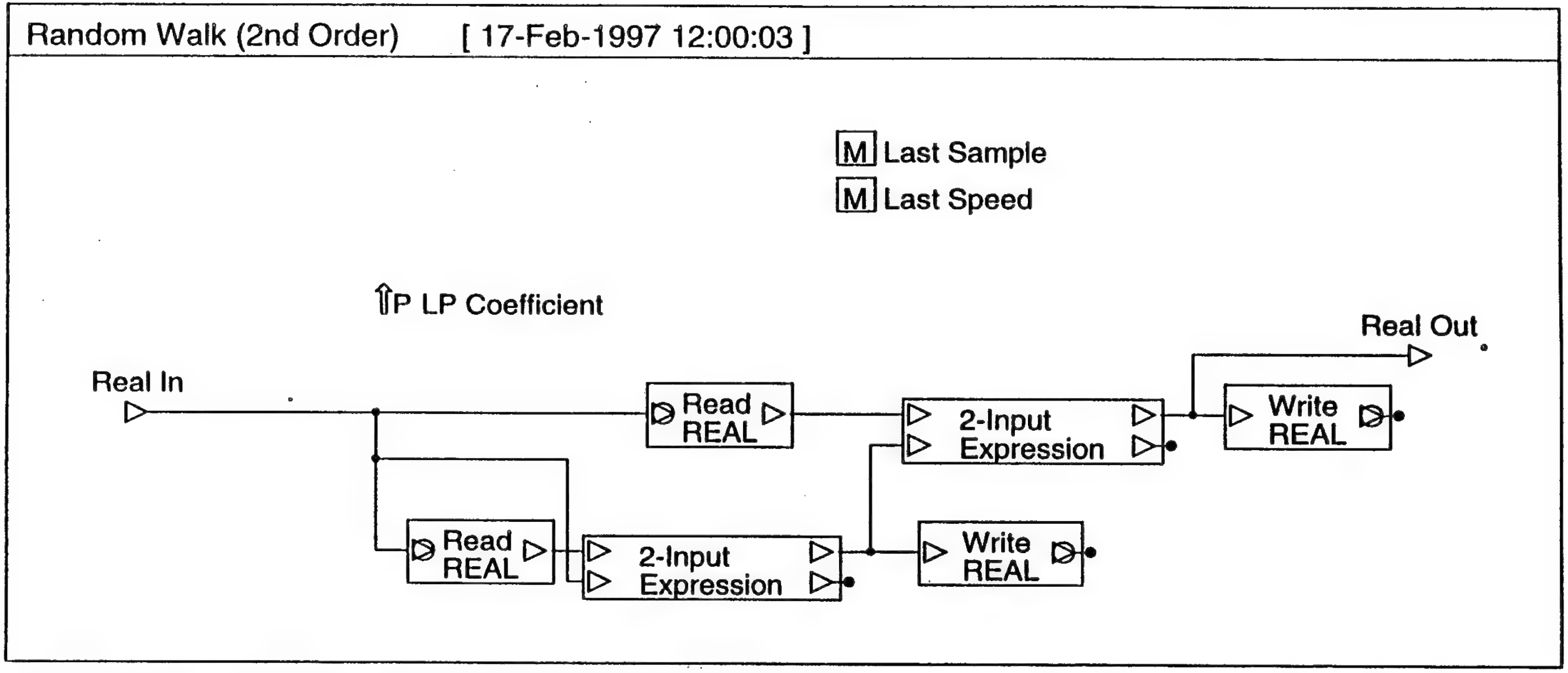
⇧P Percentage of Clock Skew

This is the Master Clock. If the a Restart Timer is never received the clock continues to function at the clock rate determined by the argument Master Clock.

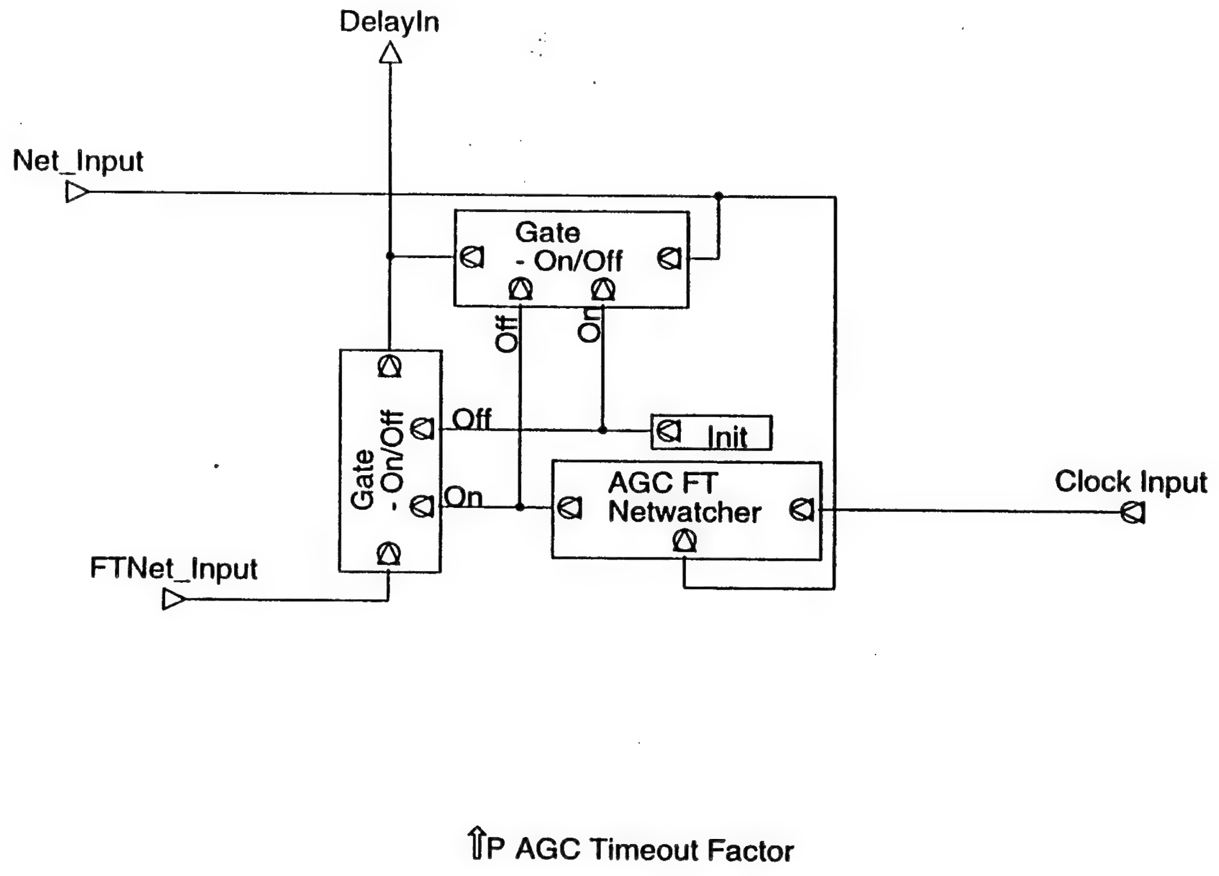
If, however, a Restart Timer is received the Master Clock will be resynced.

Pulse Time/Random Skew [17-Feb-1997 11:59:52]

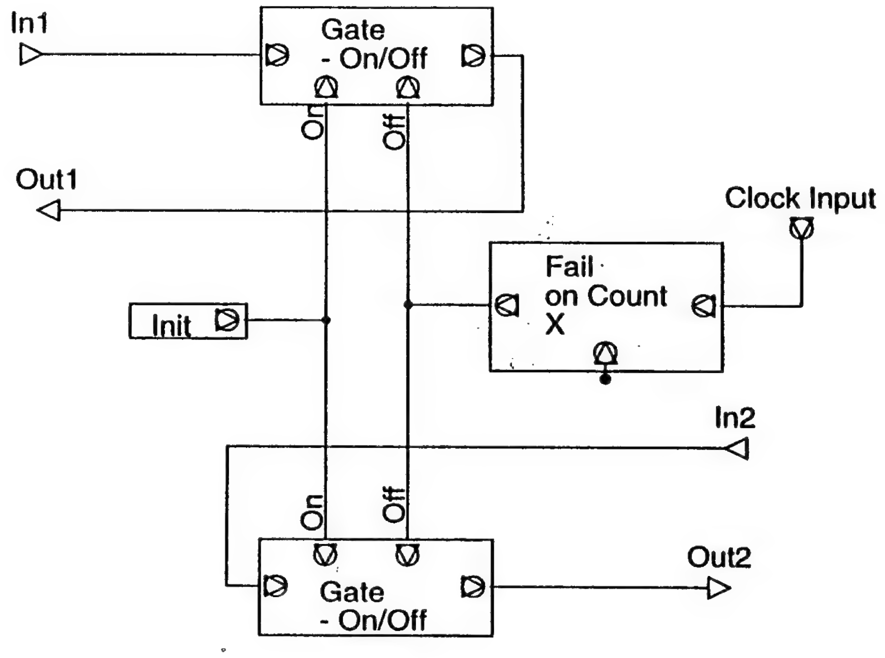



Random Walk (2nd Order) [17-Feb-1997 12:00:03]

AGC Timer [17-Feb-1997 12:00:40]

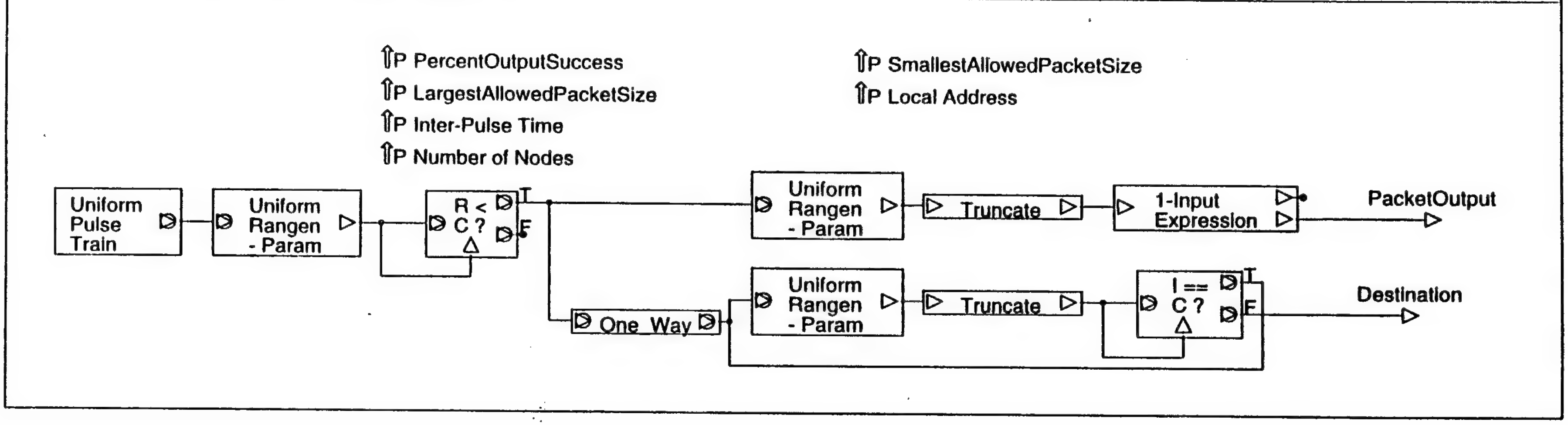


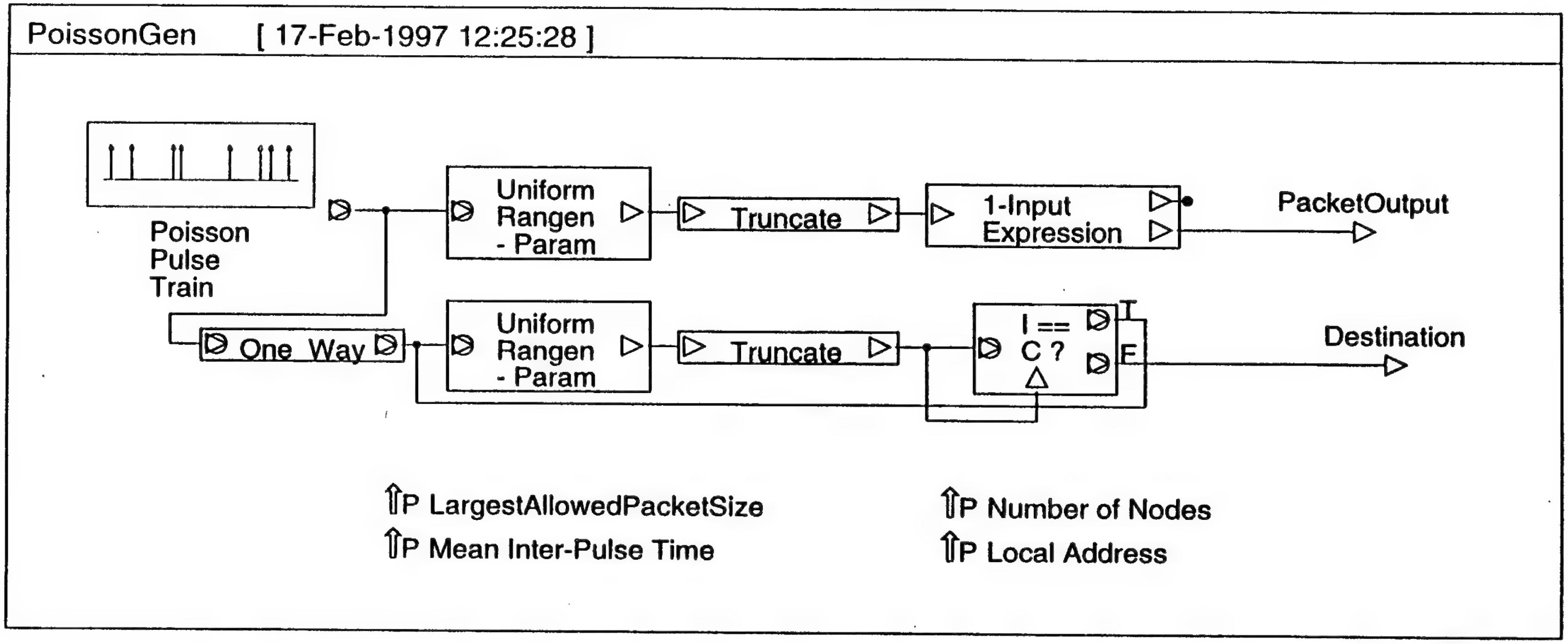
Force Failure [17-Feb-1997 12:00:51]



⇑P Fail after X counts

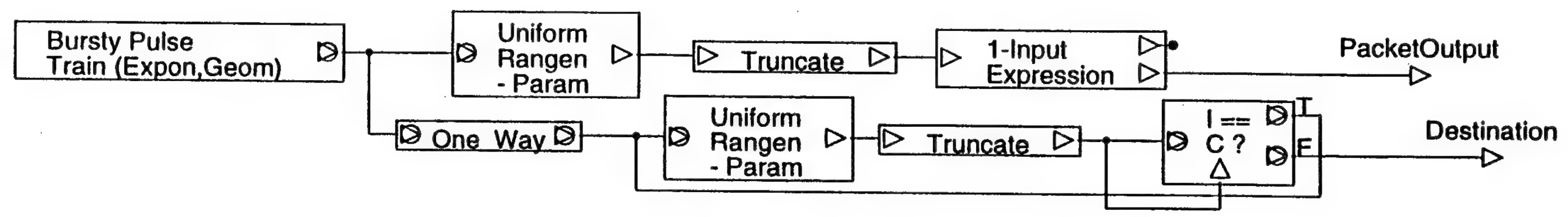
UniRandPacketGen [17-Feb-1997 12:25:13]





PoissonGen [17-Feb-1997 12:25:28]

BurstyGen [17-Feb-1997 12:25:37]



⇑P Local Address

⇑P Mean Delay Between Bursts

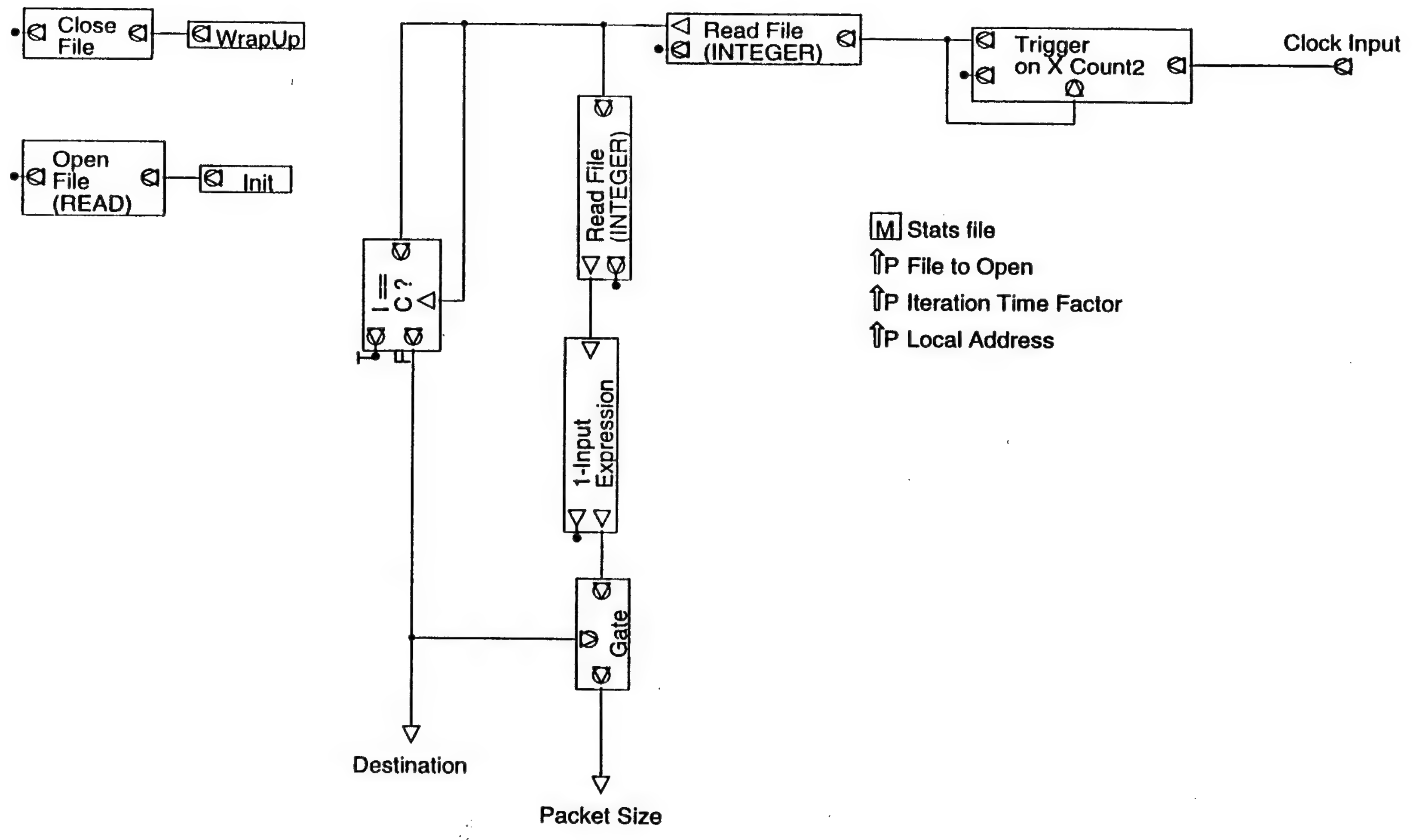
⇑P Mean number of pulses per burst

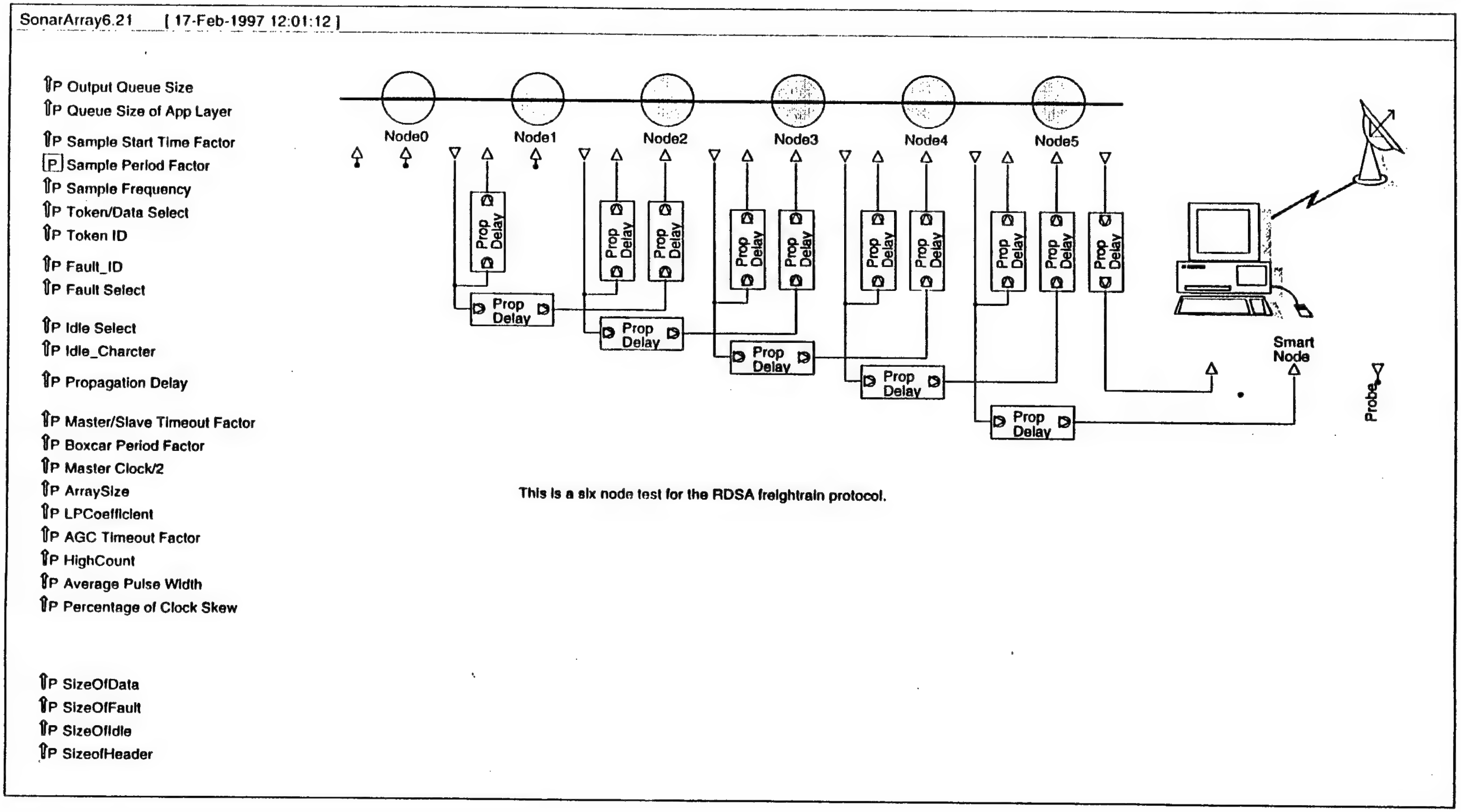
⇑P Inter-Pulse Time (during burst)

⇑P LargestAllowedPacketSize

⇑P Number of Nodes

RI Traffic Generator [17-Feb-1997 12:25:54]



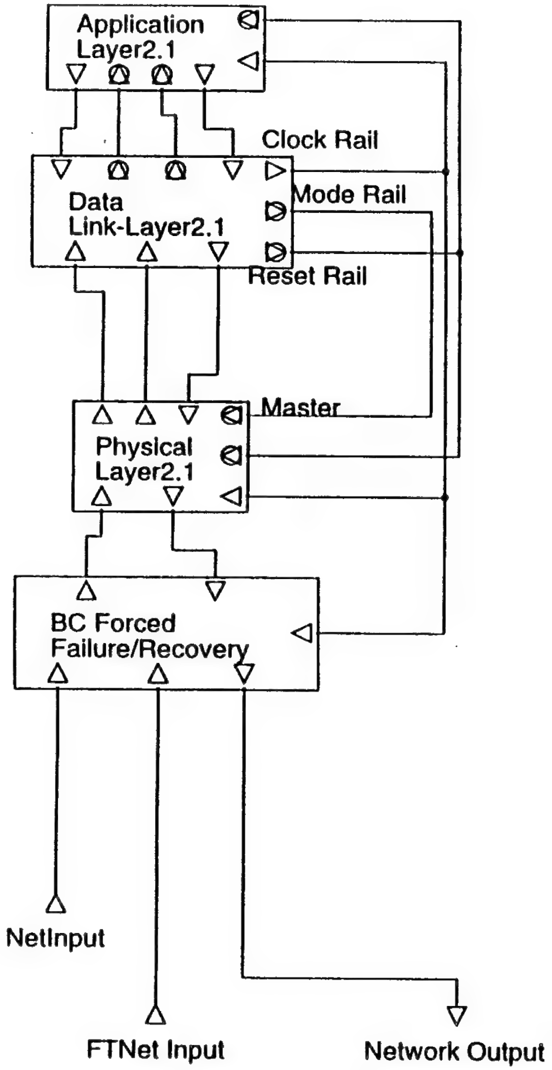


SonarArray6.21 [17-Feb-1997 12:01:12]

⇑P Output Queue Size
⇑P Queue Size of App Layer
⇑P Sample Start Time Factor
P Sample Period Factor
⇑P Sample Frequency
⇑P Token/Data Select
⇑P Token ID
⇑P Fault_ID
⇑P Fault Select
⇑P Idle Select
⇑P Idle_Charcter
⇑P Propagation Delay
⇑P Master/Slave Timeout Factor
⇑P Boxcar Period Factor
⇑P Master Clock/2
⇑P ArraySize
⇑P LPCoefficient
⇑P AGC Timeout Factor
⇑P HighCount
⇑P Average Pulse Width
⇑P Percentage of Clock Skew
⇑P SizeOfData
⇑P SizeOfFault
⇑P SizeOfIdle
⇑P SizeofHeader

This is a six node test for the RDSA freightrain protocol.

Boxcar2.1 Node [17-Feb-1997 12:01:38]



⇑P ArraySize
⇑P Master/Slave Timeout Factor
⇑P Boxcar Period Factor
⇑P Sample Start Time Factor
⇑P Sample Period Factor
⇑P Average Pulse Width
⇑P Percentage of Clock Skew
⇑P LPCoefficient
⇑P AGC Timeout Factor
⇑P HighCount

M NetFrequency
⇑P SizeOfData
⇑P SizeOfFault
⇑P SizeOfIdle
⇑P SizeofHeader

⇑P Fail after X counts
⇑P Int-Origin#

⇑P Output Queue Size
⇑P Queue Size of App Layer

⇑P Token ID
⇑P Token/Data ID

⇑P Fault_ID
⇑P Fault Select

⇑P Idle Symbol
⇑P Idle_Charcter

M Sample Time

This module is composed of the three layers which make up the Boxcar Protocol. This is the highest component layer but not the highest system layer.

Application Layer2.1 [17-Feb-1997 12:01:53]

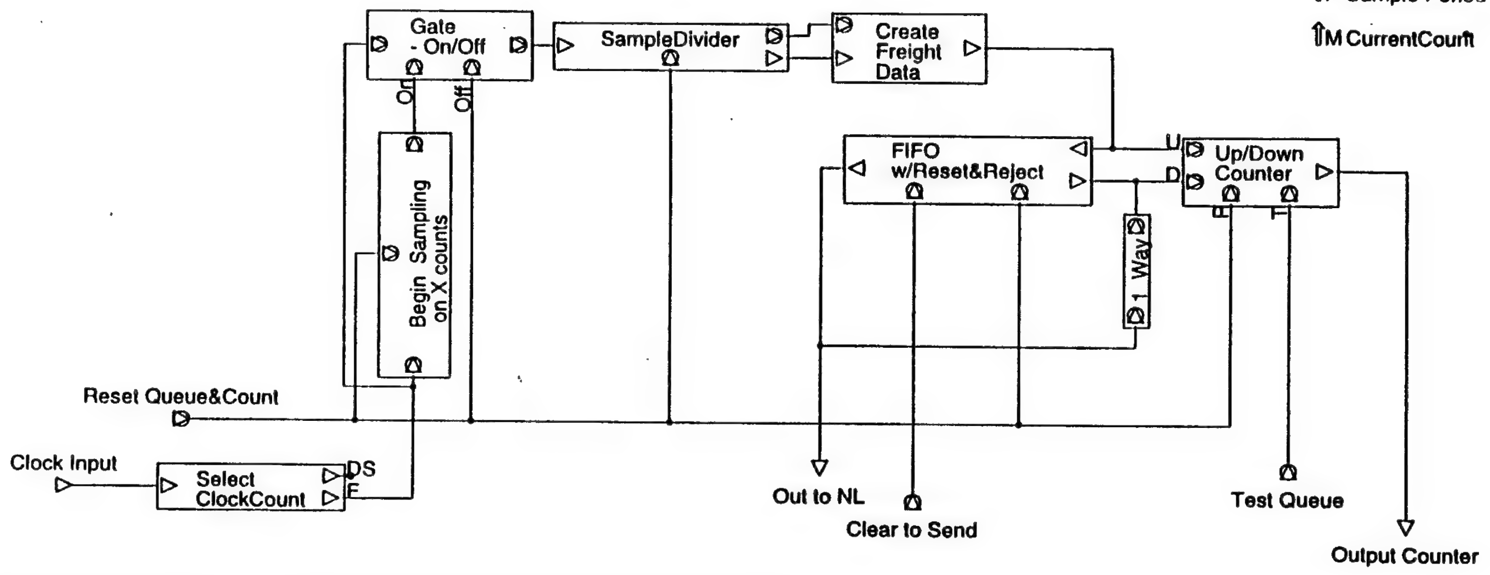
The Application Layer generates samples after X number of clocks are received from the lower Data Link Layer.

⇧P Queue Size of Session

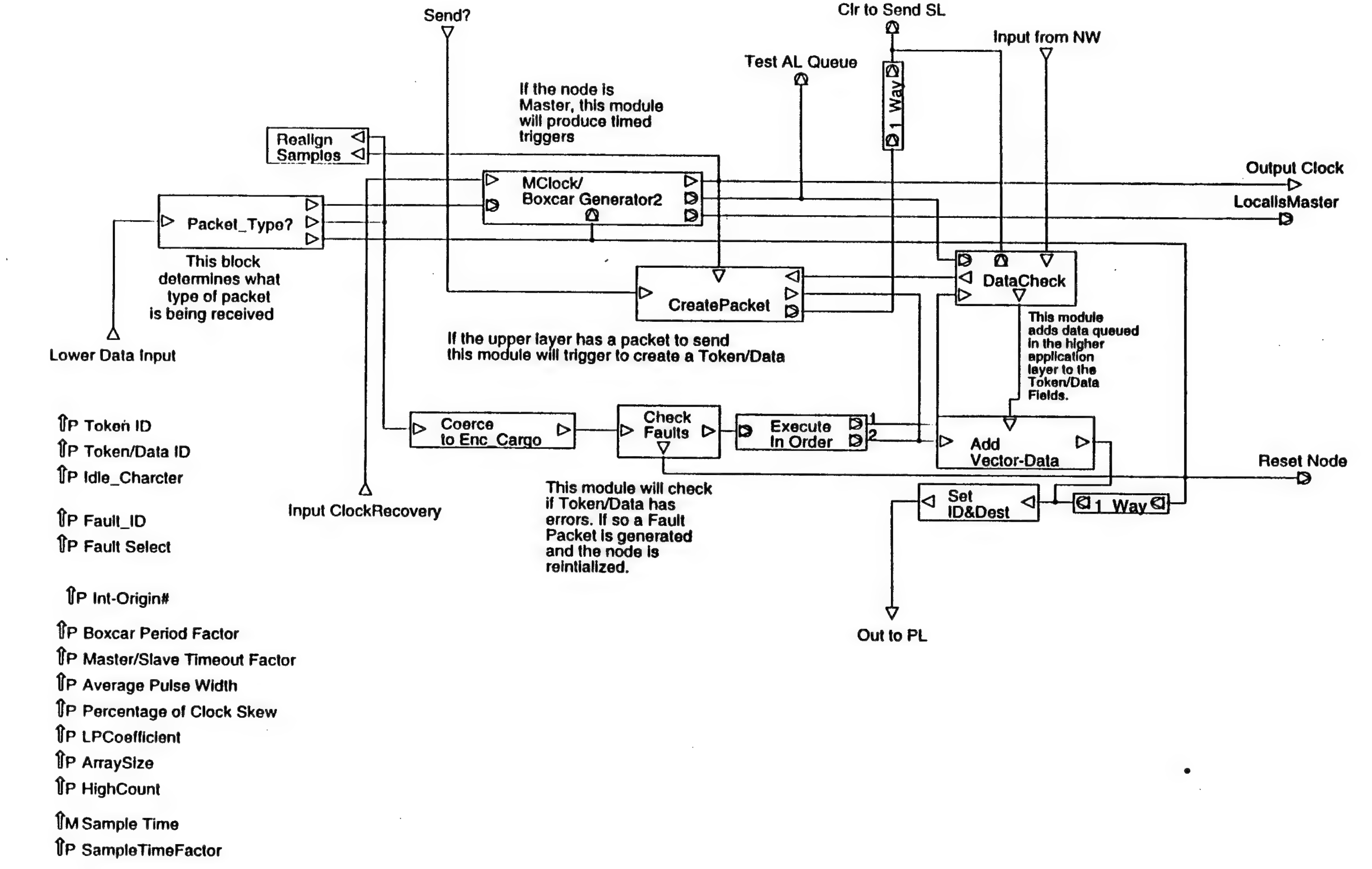
⇧P Sample Start Time Factor

⇧P Sample Period Factor

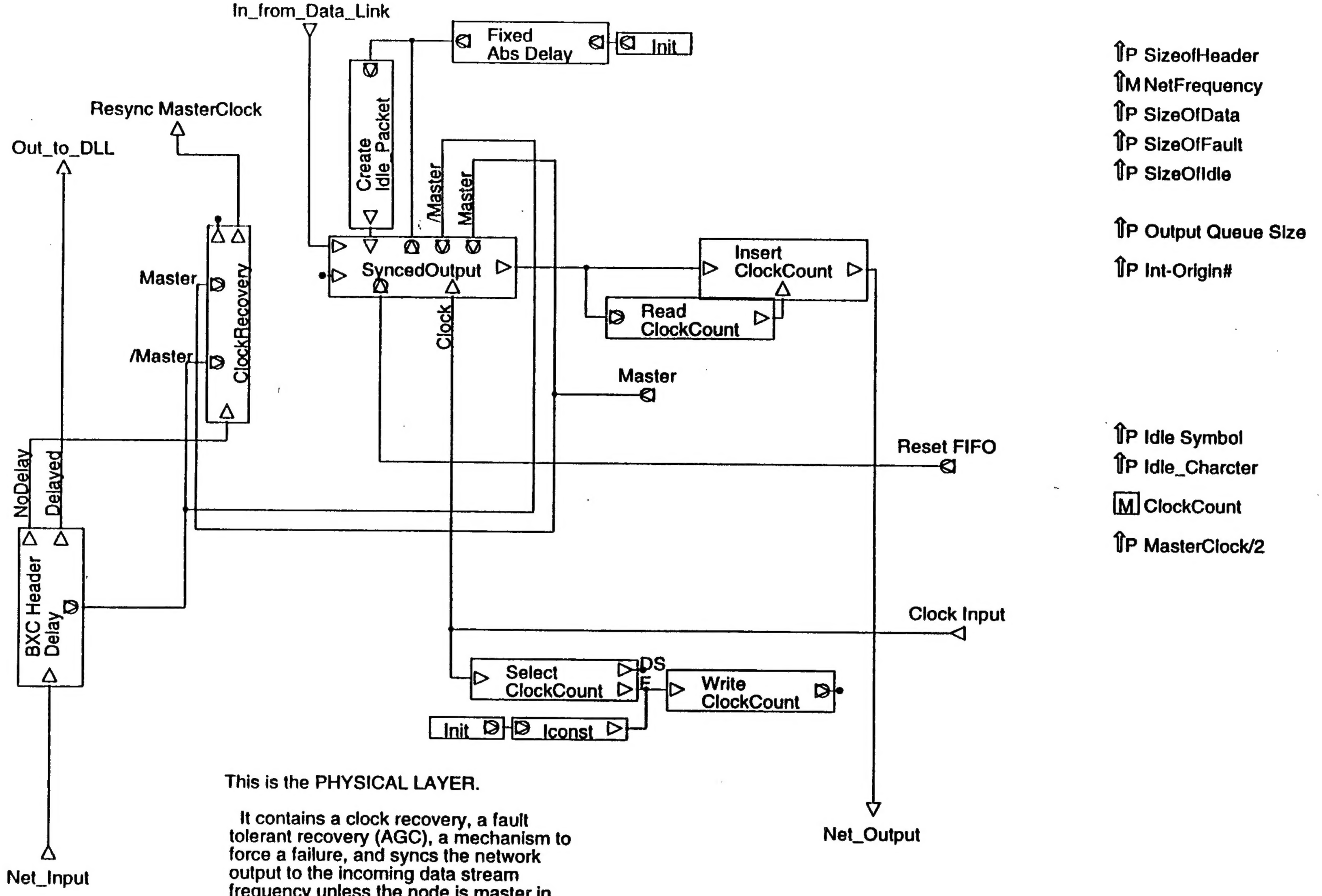
⇧M CurrentCount



Data Link-Layer2.1 [17-Feb-1997 12:02:05]


Send?
Clr to Send SL
Input from NW
Test AL Queue
1 Way
If the node is Master, this module will produce timed triggers
Realign Samples
MClock/ Boxcar Generator2
Output Clock
LocalIsMaster
Packet_Type?
This block determines what type of packet is being received
CreatePacket
DataCheck
Lower Data Input
If the upper layer has a packet to send this module will trigger to create a Token/Data
This module adds data queued in the higher application layer to the Token/Data Fields.
Coerce to Enc_Cargo
Check Faults
Execute In Order
1
2
Add Vector-Data
Reset Node
Input ClockRecovery
This module will check if Token/Data has errors. If so a Fault Packet is generated and the node is reintialized.
Set ID&Dest
1 Way
Out to PL
P Token ID
P Token/Data ID
P Idle_Charcter
P Fault_ID
P Fault Select
P Int-Origin#
P Boxcar Period Factor
P Master/Slave Timeout Factor
P Average Pulse Width
P Percentage of Clock Skew
P LPCoefficient
P ArraySize
P HighCount
M Sample Time
P SampleTimeFactor

Physical Layer2.1 [17-Feb-1997 12:02:17]



This is the PHYSICAL LAYER.

It contains a clock recovery, a fault tolerant recovery (AGC), a mechanism to force a failure, and syncs the network output to the incoming data stream frequency unless the node is master in which case the frequency of the network is determined by the master clock in the DLL.

Appendix C: Source Code

C. Source Code

All of the source code developed for this contracted research has been archived at the HCS Research Lab's world wide web site. The universal resource locater (URL) is:

http://www.hcs.ufl.edu/

The files are listed under the team name FTSA or DPSA as Source Code for Annual Report. Instructions on how to unpack and unarchive the files are included on the web page. Two versions of the source code are available. The first is a Microsoft Word 6.0 document that has been formatted to allow easy printing of the source code. The second is a *tar* file that encapsulates all of the source code as pure text which can be used for compilation.

Any questions or problems accessing these documents should be directed to Jeff Markwell at markwell@hcs.ufl.edu .

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*

2. REPORT DATE
15 Feb 97

3. REPORT TYPE AND DATES COVERED
Annual (1 Jan 96 - 31 Dec 96)

4. TITLE AND SUBTITLE
Parallel and Distributed Computing Architectures and Algorithms for Fault-Tolerant Sonar Arrays
(Annual Report #1)

5. FUNDING NUMBERS
G N00014-96-1-0569

6. AUTHORS
Alan D. George
Warren A. Rosen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
HCS Research Laboratory
FAMU-FSU College of Engineering, Florida State University
2525 Pottsdamer Street
Tallahassee, FL 32310

8. PERFORMING ORGANIZATION REPORT NUMBER

HCS-TR-97-1

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Ballston Centre Tower One
800 N Quincy Street
Arlington, VA 22217-5660

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report summarizes the progress and results of the first of a three-year study whose goal is the use of fault-tolerant distributed and parallel processing techniques to decrease the cost and improve the performance and reliability of large, disposable sonar arrays. In this first phase, tasks have concentrated on the study, design, and analysis of the fundamental components in the design and analysis of parallel and distributed computing architectures and algorithms for fault-tolerant sonar arrays as well as models for their baseline counterparts. An interactive investigation of the interdependent areas of topology, architecture, protocol, and algorithms has been conducted. The results of this first phase have helped to identify the optimum candidate network topology, architecture, hardware components, and interface protocols for the system architecture and a set of fundamental decomposition techniques and parallel algorithms for a representative set of time-domain and frequency-domain beamforming methods.

14. SUBJECT TERMS

distributed computing; parallel computing; computer networks; sonar arrays; beamforming algorithms; fault-tolerant computing

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT